Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Second release of the COMPASS Tool
Tool Grammar Reference

Deliverable Number: D31.2c

Version: 1.2

Date: January 2013

Public Document

http://www.compass-research.eu

Contributors:

Joey W. Coleman, AU

Editors:

Joey W. Coleman, AU
Stefan Hallerstede, AU

Reviewers:

Ralph Hains, Atego
Simon Foster, York

Document History

Ver	Date	Author	Description
0.1	15-10-2012	Joey Coleman	Copy D23.1 and strip it down to just the syntax rules.
0.2	08-11-2012	Joey Coleman	Start simplifying and tidying the rules, reflecting on the parser as implemented.
0.3	08-11-2012	Joey Coleman	Continue simplification of the rules.
0.4	29-12-2012	Joey Coleman	Update to the new ANTLR-based parser.
0.5	04-12-2012	Joey Coleman	Finish updating to the new ANTLR-based parser.
0.6	04-12-2012	Joey Coleman	Draft introduction of the document.
1.0	09-12-2012	Joey Coleman	Ready for internal review.
1.1	24-01-2013	Joey Coleman	Edits following internal review.
1.2	28-01-2013	Joey Coleman	Final version.

Contents

1 Introduction

The purpose of this document is to provide a reference for the grammar of the CML language, as it is accepted by the COMPASS tool. We make 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. This document does allow for several useful things:

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; and,
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 are critical for maintaining the tool in the face of changes to the CML language as the project progresses. As this document strives to be faithful to the tool, we can identify the places where new structures have been added to the language and discuss their addition in terms of the syntactic structure (as opposed to code or semantic structure).

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 write it down.

It should be noted that this document may be considered a reference for the CML_1 version of the language, as we have incorporated changes to the syntax over the original CML_0 definition, based on discussions and work in the project.

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 semicolon.

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

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

A rule definition body is a list of alternatives separated by vertical bars, |, 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.

2 CML Models and Declarations

2.1 CML Models

cml model =
    model paragraph, { model paragraph } ;

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

2.2 Value declarations

value declarations =
    'values', { value definition } ;

value definition =
    [qualifier], bindable pattern, [':', type], '=', expression ;

qualifier =
    'private'
    | 'protected'
    | 'public'
    | 'logical' ;

2.3 Channel declarations

channel declarations =
    'channels', { channel name declaration } ;

channel name declaration =
    identifier, { ',', identifier }, [':', type] ;

2.4 Chanset declarations

chanset declarations =
    'chansets', { chanset definition } ;

chanset definition =
    identifier, '=', chanset expression ;

2.5 Class declaration

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 ;

2.6 Nameset declarations

nameset declarations =
    'namesets', { nameset definition } ;

nameset definition =
    identifier, '=', nameset expression ;

2.7 State declarations

state declarations =
    'state', { instance variable definition } ;

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

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

invariant definition =
    ‘inv’, expression ;

2.8 Process declaration

process declaration =
    ‘process’, identifier, ‘=’, [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 =
    [parametrisation qualifier], identifier, { ‘,’, identifier }, ‘:’, type ;

parametrisation qualifier =
    ‘val’
    | ‘res’
    | ‘vres’ ;

2.9 Action declarations

action declarations =
    ‘actions’, { action definition } ;

action definition =
    identifier, ‘=’, [parametrisation, ‘@’], action ;

3 Processes

process =
'begin', { action paragraph } '@', action, 'end'
| process, ';', process
| process, '[]', process
| process, '|~|', process
| process, '[|', chanset expression, '|]', process
| process, '[', chanset expression, '||', chanset expression, ']', process
| process, '||', process
| process, '|||', process
| process, '/\', process
| process, '/(', expression, ')\', process
| process, '[>', process
| process, '[(', expression, ')>', process
| process, '\\', chanset expression
| process, 'startsby', expression
| process, 'endsby', expression
| '(', parametrisation, '@', process, ')',
'(', expression, { ',', expression }, ')'
| identifier, ['(', { expression }, ')']
| process, renaming expression
| ';', replication declarations, '@', process
| '[]', replication declarations, '@', process
| '|~|', replication declarations, '@', process
| '[|', chanset expression, '|]', replication declarations, '@', process
| '||', replication declarations, '@', '[', chanset expression, ']', process
| '||', replication declarations, '@', process
| '|||', replication declarations, '@', process
| '(', 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 renaming pair is used to allow a sequence of ‘.’-separated expressions. Here we allow only a single expression; consider: is `a.x.y` the identifier `a` followed by the expression `x.y`, or by two expressions, `x` and `y`? We may extend this to full sequences of ‘.’-separated expressions with restricted expressions in the future.

renaming pair =
identifier, [‘.’, expression], ‘<-’, identifier, [‘.’, expression] ;

replication declarations =
replication declaration, { ‘,’, replication declaration } ;

replication declaration =
identifier, { ‘,’, identifier }, ‘:’, type
| identifier, { ‘,’, identifier }, ‘in set’, expression
;

4 Actions

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

communication =
    identifier, { communication parameter } ;

communication parameter =
    '?', bindable pattern, ['in set', expression]
    | '!', parameter
    | '.', parameter ;

parameter =

identifier
| '(' expression ')'
| symbolic literal
| tuple expression
| record expression ;

parallel action =

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

parametrised action =
'(' parametrisation, { ',', parametrisation }, '@', action ')' ;

instantiated action =
parametrised action, '(', expression, { ',', expression }, ')' ;

replicated action =

';', replication declarations, '@', action
| '[]', replication declarations, '@', action
| '|~|', replication declarations, '@', action
| '[||', nameset expression, '||]', replication declarations, '@', action
| '|||', replication declarations, '@', '[', nameset expression, ']', action
| '[|', chanset expression '|]', replication declarations, '@',
'[' , nameset expression, ']', action
| '||', replication declarations, '@',
'[', nameset expression, '|', chanset expression, ']', action
| '||', replication declarations, '@', '[', nameset expression, ']', action
;

5 Statements

statement =
'`let`', local definition, { '`,`', local definition }, '`in`', action
| '`(`', [block declarations], action, '`)`'
| cases statement
| if statement
| '`if`' non-deterministic alt, { '`|`', non-deterministic alt }, '`end`'
| '`do`' non-deterministic alt, { '`|`', non-deterministic alt }, '`end`'
| '`while`', expression, '`do`', action
| '`for`', bindable pattern, ['`:`', type] '`in`', expression, '`do`', action
| '`for`', '`all`', bindable pattern, '`in set`', expression, '`do`', action
| '`for`', identifier, '`=`', expression, '`to`', expression, ['`by`', expression],
'`do`', action
| '`[`', [frame], ['`pre`', expression], '`post`', expression, '`]`'
| '`return`', [expression]
| assign statement
| multiple assign statement
| call statement
| new statement ;

local definition =
value definition
| function definition ;

block declarations =
'`dcl`', assignment definition, { '`,`', assignment definition }, '`@`' ;

non-deterministic alt =
expression, '`->`', action ;

5.1 Conditional Statements

Note that the syntax described here suffers from the "dangling-else" ambiguity that affect many C-like languages. We resolve this in the parser in a manner similar to those C-like langauges, so that the else and elseif clauses attach to the nearest if. For example, "if a then if b then skip else skip" would be correctly parenthesised as "if a then (if b then skip else skip)".

if statement =

'`if`', expression, '`then`', action, { elseif statement }, ['`else`', action] ;

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

cases statement =
    ‘cases’, expression, ‘:’,
      cases statement alt, { ‘,’, cases statement alt }
      [‘,’, others statement],
    ‘end’ ;

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

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

5.2 Assignment-like Statements

assign statement =
    assignable expression, ‘:=’, expression ;

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

Note that the second alternative of the call statement and the basic assign statement partially overlap and introduce an ambiguity that is resolved by the typechecker.

call statement =

name, ‘(’, [expression, { ‘,’, expression }], ‘)’
| assignable expression, ‘:=’, name, ‘(’, [expression, { ‘,’, expression }], ‘)’ ;

new statement =

assignable expression, ‘:=’, ‘new’, name, ‘(’, [expression, { ‘,’, expression }], ‘)’ ;

6 Expressions

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

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

old name =
    identifier, '~' ;

unary operator =

‘+’ | ‘-’ | ‘abs’ | ‘floor’ | ‘not’ | ‘card’ | ‘power’ | ‘dunion’ | ‘dinter’ |
‘hd’ | ‘tl’ | ‘len’ | ‘elems’ | ‘inds’ | ‘reverse’ | ‘conc’ | ‘dom’ | ‘rng’ |
‘merge’ | ‘inverse’ ;

binary operator =
‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘div’ | ‘rem’ | ‘mod’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘=’ |
‘<>’ | ‘or’ | ‘and’ | ‘=>’ | ‘<=>’ | ‘in set’ | ‘not in set’ | ‘subset’ |
‘psubset’ | ‘union’ | ‘\’ | ‘inter’ | ‘^’ | ‘++’ | ‘munion’ |
‘<:’ | ‘<-:’ | ‘:>’ | ‘:->’ | ‘comp’ | ‘**’ ;

tuple expression =
‘mk_’, ‘(’, expression, ‘,’, expression, { ‘,’, expression }, ‘)’ ;

record expression =
‘mk_’, ‘token’, ‘(’, expression, ‘)’
| ‘mk_’, name, ‘(’, [expression, { ‘,’, expression }], ‘)’ ;

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

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

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

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

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

apply =
expression, ‘(’, [expression, { ‘,’, expression }], ‘)’ ;

field select =
expression, ‘.’, identifier ;

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

6.1 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 =
    pattern list, '->', expression, { ',', pattern list, '->', expression } ;

6.2 Assignable Expressions

assignable expression =
    'self' { selector }
    | identifier { selector } ;

selector =
    '(', [expression, { ',', expression }], ')'
    | '(', expression, '...', expression, ')'
    | '.#', numeral
    | '.', identifier ;

7 Functions

function declarations =
    ‘`functions`’, { 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] ;

parameters list =
    parameters, { parameters } ;

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

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

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

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

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

8 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 =
    '`operations`', { 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`' ;

implicit operation definition =
    [qualifier], identifier, parameter types, [identifier type pair list],
    [frame],
    ['`pre`', expression],
    '`post`', expression ;

frame =
    '`frame`', var information, { var information } ;

var information =
    '`rd`', name, { '`,`', name }, ['`:`', type]
    | '`wr`', name, { '`,`', name }, ['`:`', type] ;

9 Chanset and Nameset Expressions

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

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

Note that the third alternative –the comprehension– here only allows an optional single expression, unlike the sequence allowed in CML_0; this restriction is due to the same ambiguities with '.' that affect the restrictions in the renaming pair rule.

The structure of the nameset expression rule is identical to that of the chanset expression rule (and, indeed, the parser in the tool does not distinguish between them in recognition).

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

10 Patterns and Bindings

10.1 Patterns

pattern =
    bindable pattern
    | match value ;

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

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

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

10.2 Bindings

bind =
    set bind | type bind ;

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

type bind =
    pattern, ':', type ;

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

multiple bind =
    pattern list, 'in set', expression
    | pattern list, ':', type ;

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

11 Types

type declarations =
'types', type definition, { ';', type definition } ;

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

type =
'(', type, ')'
| basic type
| quote literal
| 'compose', identifier, 'of', field list, 'end'
| type, '|', type, { '|', type }
| type, '*', type, { '*', type }
| '[', type, ']'
| 'set of', type
| 'seq of', type
| 'seq1 of', type
| 'map', type, 'to', type
| 'inmap', type, 'to', type
| function type
| name ;

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

field list =
field, { field } ;

field =
type
| identifier, ':', type
| identifier, ':-', type ;

function type =
discretionary type, '+>', type
| discretionary type, '->', type ;

discretionary type =
type
| '()' ;

type invariant =

‘`inv`’, pattern, ‘`==`’, expression ;

12 Lexical Specification

Unlike the rest of this specification, the rules in this section are sensitive to whitespace; as such, whitespace may not implicity 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 =
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 =
Any character in the ranges [U+0041,U+005A] and [U+0061,U+007A] (A-Z and a-z, respectively)

character =
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 =
initial letter, { following letter } ;

digit =
‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ ;

hex digit =
digit | ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ ;

numeral =
digit, { digit } ;

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

numeric literal =
    decimal literal
    | hex literal ;

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

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

hex literal =
    (‘0x’ | ‘0X’), hex digit, { hex digit } ;

boolean literal =
    ‘true’ | ‘false’ ;

nil literal =
    ‘nil’ ;

character literal =
    ‘’’, character, ‘’’
    | ‘’’, escape sequence, ‘’’ ;

escape sequence =
    ‘\\’ | ‘\r’ | ‘\n’ | ‘\t’ | ‘\f’ | ‘\e’ | ‘\a’ | ‘\"’ | ‘\’’
    | ‘\x’, hex digit, hex digit
    | ‘\u’, hex digit, hex digit, hex digit, hex digit
    | ‘\c’, ascii letter ;

text literal =
    ‘"’, { character | escape sequence }, ‘"’ ;

quote literal =
    ‘<’, identifier, ‘>’ ;

Index of Rule Definitions