Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Third release of the COMPASS Tool
CML Grammar Reference

Deliverable Number: D31.3c

Version: 1.0

Date: November 2013

Public Document

http://www.compass-research.eu

Contributors:

Joey W. Coleman, AU

Editors:

Joey W. Coleman, AU

Reviewers:

Document History

Ver	Date	Author	Description
0.1	06-11-2013	JWC	Initial version based on cml-syntax wiki version #5c37827
0.2	08-11-2013	JWC	Update prefatory text, include "What's new" section
1.0	26-11-2013	JWC	Final version

Contents

1 Introduction

The purpose of this document is to provide a reference for the grammar of the COMPASS 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 [BCW13].

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; 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.

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

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, each 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.

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

Figure 2: An example grammar rule.

with the name of the rule, then an equality symbol, =, then the rule definition body, then a semicolon.

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.

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.

1.2 Major changes relative to previous versions

The next release of the Symphony IDE–this document accompanies version 0.2.2– will introduce support for reading configuration blocks in CML models, and we have included necessary grammar elements in this document. Configuration blocks allow a single set of files to specify multiple SOS models; and this feature anticipates changes that will be necessary for the forthcoming refinement plugin.

See Section 4 and the rules *value declarations*, *channel declarations*, *chanset declarations*, *type declarations*, and *function declarations*. Note that all syntax changes for configuration blocks are optional, and do not affect the compatibility of previous models.

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*
  | *configuration block*
  ;

3 Declarations

value declarations →
  'values', ['(', *identifier*, ')'], { *value definition* }
  ;

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

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

channel declarations →
  'channels', ['(', *identifier*, ')'], { *channel name declarations* }
  ;

channel name declarations →
  identifier, { ',', *identifier* }, [':', *type*]
  ;

chanset declarations →
  'chansets', ['(', *identifier*, ')'], { *chanset definition* }
  ;

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

nameset declarations →
  'namesets', { *nameset definition* }
  ;

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

state declarations →
  'state', { *instance variable definition* }
  ;

instance variable definition →
  [*qualifier*], *assignment definition*
  | *invariant definition*
  ;

assignment definition →
  identifier, ':', *type*, [*assignment definition value*]
  ;

assignment definition value →
  ':=', *expression*
  | 'in', *expression*
  ;

invariant definition →
  'inv', *expression*
  ;

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`’
;

action declarations →
‘`actions`’, { *action definition* }
;

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

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*
;

nameset expression →
chanset expression
;

4 Configuration Blocks

configuration block →
'configuration', *identifier*, 'includes', *identifier*, { ',', *identifier* }, 'end'
;

5 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*
;

6 Processes

process →
action process
| *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*, { ',', *expression* }], ')']

| *process*, *renaming expression*
| *replicated process*
| '(', *process*, ')'
;

action process →
'begin', { *action paragraph* }, '@', *action*, 'end'
;

replicated process →
';', *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*
;

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*
;

7 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*, ‘startsby’, *expression*
| *action*, ‘endsby’, *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*, [‘:’, ‘(’, *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*
;

8 Statements

statement →
'let', *local definition*, { ',', *local definition* }, 'in', *action*
| '(', ['dcl', *assignment definition*, { ',', *assignment definition* }, '@'], *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*
;

non-deterministic alt →
expression, '->', *action*
;

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*
;

assign statement →
assignable expression, ':=', *expression*
;

multiple assign statement →
'atomic', '(', *assign statement*, ';', *assign statement*, { ';', *assign statement* }, ')'
;

call statement →
name, '(', [*expression*, { ',', *expression* }], ')'
| *assignable expression*, ':=', *name*, '(', [*expression*, { ',', *expression* }], ')'
;

new statement →
assignable expression, ':=', 'new', *name*, '(', [*expression*, { ',', *expression* }], ')'
;

9 Types

type declarations →
'types', ['(', *identifier*, ')'], [*type definition*, { ';', *type definition* }]
;

type definition →
[*qualifier*], *identifier*, '=', *type*, [*type invariant*]
| [*qualifier*], *identifier*, '::', { *field* }, [*type invariant*] }
;

type →
'(', *type*, ')'
| *basic type*
| *quote literal*
| 'compose', *identifier*, 'of', { *field* }, '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 →
type
| *identifier*, ':', *type*
| *identifier*, ':-', *type*
;

function type →
discretionary type, '+>', *type*

| *discretionary type*, '->', *type*
;

discretionary type →
type | '()'
;

type invariant →
'inv', *pattern*, '==', *expression*
;

10 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*]
;

11 Functions

function declarations →
'functions', ['(', *identifier*, ')'], { *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’
;

12 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*
;

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* }
;

assignable expression →
'self' { *selector* }
| *identifier* { *selector* }
;

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

13 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* }
;

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* }
;

14 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 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*, ‘>’
;

References

[BCW13] Jeremy Bryans, Samuel Canham, and Jim Woodcock. CML Definition 3 — Denotational Semantics. Technical report, COMPASS Deliverable, D23.4, September 2013. Available at http://www.compass-research.eu/.

Index of Rule Definitions