

CML2 syntax (2.2.1)

Anjo Anjewierden and Guus Schreiber
SWI, University of Amsterdam
Roetersstraat 15
1018 WB Amsterdam
email: `anjo@swi.psy.uva.nl`

August 14, 2003

Abstract

This document contains the syntax of CML2. CML (Conceptual Modelling Language) was originally developed as part of CommonKADS. CML2 takes into account practical usage of the original CML language.

Also included is a description of a parser for CML2. The CML parser is a stand-alone program capable of checking the syntax of COMMONKADS CML2. The parser has an option to generate output for other programs.¹

¹This work was partially supported by the European Communities Esprit programme and the Netherlands Computer Science Research Foundation with financial support from the Netherlands Organization for Scientific Research (NWO).

Contents

1	Introduction	4
1.1	Tool support	4
1.2	Release history	4
1.2.1	Changes between CML 2.0 and 2.1	4
1.2.2	Changes between CML 2.1 and 2.2	4
1.3	Acknowledgements	5
2	Overview	5
2.1	Syntactic conventions	5
2.2	Low-level syntax	5
2.2.1	Comments	5
2.2.2	Names	5
2.2.3	Hyphens, underbars and spaces	6
2.3	Operators in expressions	6
3	Design	7
3.1	Ordering	8
3.2	Incompleteness	8
3.3	Unresolved names	9
4	Syntax	9
4.1	Synonyms	9
4.2	Knowledge model	9
4.3	Domain knowledge	10
4.3.1	Domain schema	10
4.3.2	Concept	10
4.3.3	Axioms	11
4.3.4	Attributes	11
4.3.5	Rule type	12
4.3.6	Mathematical model	12
4.3.7	Structure	13
4.3.8	Relation	13
4.3.9	Binary relation	14
4.3.10	Type range	14

4.3.11 Value type	14
4.4 Knowledge base	15
4.4.1 Instances	15
4.5 Inference knowledge	18
4.5.1 Inference	18
4.5.2 Transfer function	18
4.5.3 Knowledge role	18
4.6 Task knowledge	19
4.6.1 Task	19
4.6.2 Task method	19
4.7 Control structure	19
4.8 PSM knowledge	20
4.8.1 PSM	21
4.9 Ontology mapping	21
4.10 Equations	21
4.10.1 Equation	22
4.10.2 Operators	22
4.11 Support	22
4.11.1 Cardinality	22
4.11.2 Role	23
4.11.3 Terminology	23
5 Parser	23
5.1 Disclaimer	23
5.2 Syntactic units	24
5.2.1 Names of keywords	24
5.2.2 Completeness	24
5.2.3 Names of constructs	24
5.2.4 Unresolved constructs	25
5.3 Options	25
5.4 Examples	27
5.5 Error recovery	28
5.6 Error messages	28

1 Introduction

CML2 (Conceptual Modelling Language) is a semi-formal notation for the specification of COMMONKADS knowledge models. See <http://www.commonkads.uva.nl> for details.

Parts of CML2 can be understood and used independently of COMMONKADS, in particular the CML2 notation for domain knowledge.

1.1 Tool support

CML2, as documented here, is supported by a set of tools. The most useful tool is the CML parser. The CML parser takes a file containing CML2 and checks the contents for (syntactic) correctness. The parser can also generate an intermediate representation (called ECO). The intermediate representation is syntactically much simpler than CML2 and it is therefore much easier to interpret by computer programs. Documentation of the CML parser can be found in Chapter 5.

Another tool is KADS22. This is an interactive environment in which the user can browse, manipulate and generate CML2. The CML parser and KADS22 are tightly integrated.

The tools can be downloaded from the following website:

<http://www.swi.psy.uva.nl/projects/kads22>

This site also contains the latest version of the documentation, including the documentation you are reading now.

1.2 Release history

The syntax is supposed to remain stable for the parts of CML2 that are frequently used and widely understood.

1.2.1 Changes between CML 2.0 and 2.1

Name changes

The names of several constructs have been changed because the old names were too difficult for students (yes, this is the best explanation we found so far!). The old names still work, see page 9 for a list of all changes.

1.2.2 Changes between CML 2.1 and 2.2

Image as a primitive type

A relatively minor change is the addition of **image** as a primitive-type (page 14) for attributes. The intended value of such attributes is a location (file, URL) which contains a displayable image. This change is motivated by the desire to decorate constructs with multi-media.

Syntax for instances

The syntax for instances is now similar to the syntax for the constructs they are instances of. A description with some examples can be found on page 15. This change is motivated by the desire to make the syntax as internally consistent as feasible.

1.3 Acknowledgements

Comments by Richard Benjamins, Jan Wielemaker, Alexander Boer, Bob Wielinga and Robert de Hoog are gratefully acknowledged.

2 Overview

2.1 Syntactic conventions

The following conventions are used in the syntax specification:

$X ::= Y$	The syntax of X (a non-terminal) is defined by Y .
$[X]$	Zero or one occurrence of X .
X^*	Zero or more occurrences of X .
X^+	One or more occurrences of X .
$X Y \dots$	One or more occurrences of X separated by Y . This construct is mainly used to abbreviate comma-separated lists. For example, “Name, ...” is short for “Name $\langle, \text{Name} \rangle^*$ ”.
$X Y$	One of X or Y (exclusive or).
$\langle X \rangle$	Grouping construct for specifying the scope of operators.
symbol	Bold: predefined terminal symbols of the language. In the syntax definition these symbols are given in lower case. In a CML2 file they must be given in upper case.
<i>Symbol</i>	Capitalised: user defined terminal symbols of the language.
symbol	Lowercase: non-terminal symbols.
”Text”	Arbitrary text between double quotes. A double quote inside the text can be escaped with a backslash.
‘X’	Escapes the operator symbol (e.g. $*$) and denotes the literal X .

2.2 Low-level syntax

2.2.1 Comments

Comments are not formerly part of the syntax of CML2. Comments follow the C style: initiate a comment with $/ *$ and terminate with $*/$. Comments can appear anywhere between the symbols of the CML2 syntax.

2.2.2 Names

The most frequently occurring low-level construct is a **name**. CML2 defines a name to start with a letter followed by letters, digits, hyphens and underbars in an arbitrary order. If additional characters

are required in a name, the entire name must be embedded in single quotes. For example, the concept *Monster of Loch Ness* is defined as follows:

```
CONCEPT 'Monster of Loch Ness';  
...  
END CONCEPT 'Monster of Loch Ness';
```

What a name refers to is suggested by the convention *Construct*-name, where *Construct* is the type of thing the name denotes. The CML parser will take this rather literally, if the parser encounters a *Concept*-name, it will create a concept of the given name if it did not already exist.

2.2.3 Hyphens, underbars and spaces

The ASCII character set has given us two symbols that are used interchangeably: the hyphen and the underbar. Some languages allow a hyphen in a name (e.g. Lisp), whereas others disallow it (e.g. C). CML2 allows both characters as it is a language in which one may want to denote concepts which already have an established notation (e.g. *hole-in-1*).

Users of CML2 are advised to use the notation of a concept as it appears in (public) sources. In general, the consequence is to use hyphens and spaces rather than underbars. To avoid practical problems when CML2 is translated to other languages, the CML parser has several options through which hyphens and spaces can be converted to underbars automatically.

2.3 Operators in expressions

The table below lists the operators that can be used in expressions. Note that the operator for equality is == (and not =). The main entry point in the syntax for expressions is equation (page 22).

Operators are listed in order of increasing precedence. Operators of equal precedence are grouped between horizontal rulers.

Operator	Description
=	Equivalence (mathematics)
:=	Assignment (programming)
<	Less than (comparison)
<=	Less or equal (comparison)
>	More than (comparison)
>=	More or equal (comparison)
==	Equal (comparison)
!=	Not equal (comparison)
->	Implication (logical)
<-	Inverse implication (logical)
<->	Double implication (logical)
AND	Conjunction (logical)
OR	Disjunction (logical)
XOR	Choice (logical)
+	Addition (arithmetic)
-	Subtraction (arithmetic)
*	Multiplication (arithmetic)
/	Division (arithmetic)
**	Exponentiation (arithmetic)
-	Negation (arithmetic)
NOT	Negation (logical)
.	Dereference (programming)
`	Derivative (mathematics)
(...)	Grouping
[...]	Subscript

Note that a hyphen may be used in names and is also an operator. White space around hyphens intended as a minus sign may thus be necessary.

```

CONCEPT vehicle;
  ATTRIBUTES:
    no-of-wheels: INTEGER;

  AXIOMS:
    no-of-wheels > 3;
END CONCEPT vehicle;

```

The hyphens part of `no-of-wheels` are not minus signs. Obviously, `no - of - wheels` is something completely different.

3 Design

This section contains a number of considerations for the design of CML2.

The two most important practical problems with CML2 are the highly elaborate syntax (about 170 keywords special symbols) and the fact that users initially want to be informal and sloppy.

To alleviate some of the practical problems, there is a difference between the syntax definition and how CML2 parsers should behave. The sections below address this.

3.1 Ordering

The ordering of attributes of a construct is more or less arbitrary. Consider the following two samples of CML2:

```
CONCEPT vehicle;
  ATTRIBUTES:
    wheels: INTEGER;
  AXIOMS:
    wheels > 3;
END CONCEPT vehicle;
```

versus

```
CONCEPT vehicle;
  AXIOMS:
    wheels > 3;
  ATTRIBUTES:
    wheels: INTEGER;
END CONCEPT vehicle;
```

In these examples the only difference is that **attributes** and **axioms** are reversed. The syntax definition states that **attributes** come before **axioms** (there is some logic for this as the example shows). It is, however, often difficult to remember the precise order, and therefore both of the above examples are considered to have the same meaning.

3.2 Incompleteness

A definition of a construct may be incomplete. Suppose you know that the domain contains a binary-relation named **consists-of**. The syntax states that the relation has at least two attributes (the types of the arguments). A parser will accept an incomplete definition like the following:

```
BINARY-RELATION consists-of;
END BINARY-RELATION consists-of;
```

Allowing this makes it possible to create an *inventory* of the domain by just listing the names of constructs and their type.

3.3 Unresolved names

A definition may contain references to undefined entities. This makes incremental development a lot easier. For example:

```
CONCEPT human;
  SUPER-TYPE-OF: man, woman;
END CONCEPT human;
```

is acceptable without defining **man** and **woman**. See also `-unresolved`.

4 Syntax

4.1 Synonyms

The following table lists the synonyms of terms used in the CML2 syntax specification. The term in the left column is used in this document, the term(s) in the right column give synonyms used in previous versions of CML2 and are still accepted by the CML parser.

Term	Old version (still valid)
attributes	properties
domain-schema	ontology, domain-knowledge-schema
knowledge-base	domain-model
knowledge-model	expertise-model
list-of	listof, list
rule-type	rule-schema
set-of	setof, set
specification	spec
use	uses, import
value-specification	value-spec
value-type	value-set

4.2 Knowledge model

```
knowledge-model ::= knowledge-model Knowledge-model ;
                  [ terminology ]
                  domain-knowledge
                  inference-knowledge
                  task-knowledge
                  [ psm-knowledge ]
                  end knowledge-model [ Knowledge-model ; ] .
```

It is usual for **psm knowledge** to be defined separately, for example as part of a library of PSMs.

4.3 Domain knowledge

```

domain-knowledge      ::= domain-knowledge Domain-knowledge ;
                        [ terminology ]
                        ⟨ domain-schema |
                        ontology-mapping | knowledge-base ⟩ *
                        end domain-knowledge [ Domain-knowledge ; ] .

```

4.3.1 Domain schema

A domain-schema is defined through the specification of *types* or *constructs*. CML2 provides several representational primitives: concept (page 10), relation (page 13), rule-type (page 12) and structure (page 13).

The keyword **definitions**, to introduce the constructs defined in the schema, is no longer required.

```

domain-schema        ::= domain-schema Domain-schema ;
                        [ terminology ]
                        [ use : use-construct , ... ; ]
                        [ definitions : ] domain-construct*
                        end domain-schema [ Domain-schema ; ] .

use-construct        ::= Domain-schema |
                        Construct from Domain-schema .

domain-construct     ::= binary-relation | concept | mathematical-model |
                        relation | rule-type | structure | value-type .

```

4.3.2 Concept

The notion of *concept* is used to represent a class of real or mental objects in the domain being studied. The term concept corresponds roughly to the term *entity* in ER-modelling and *class* in object-oriented approaches.

Every concept has a *name*, a unique symbol which can serve as an identifier of the concept, possible super concepts (multiple inheritance is allowed).

```

concept              ::= concept Concept ;
                        [ terminology ]
                        [ super-type-of : Concept , ... ;
                        [ disjoint : yes | no ; ]
                        [ complete : yes | no ; ] ]
                        [ sub-type-of : Concept , ... ; ]
                        [ has-parts : has-part+ ]
                        [ part-of : Concept , ... ; ]
                        [ viewpoints : viewpoint+ ]
                        [ attributes ]
                        [ axioms ]
                        end concept [ Concept ; ] .

```

```

has-part          ::= Concept ;
                  [ role ]
                  [ cardinality ] .

viewpoint         ::= dimension :
                  Concept , ... ;
                  [ disjoint : yes | no ; ]
                  [ complete : yes | no ; ] .

```

4.3.3 Axioms

The axioms slot supports the specification of (mathematical) relationships that are defined to be true.

```

axioms            ::= axioms :
                  equation ; ... .

```

Examples. Consider the definition of a chess-square:

```

CONCEPT chess-square;
  ATTRIBUTES:
    rank: INTEGER;
    file: INTEGER;

  AXIOMS:
    1 >= rank >= 8;
    1 >= file >= 8;
END CONCEPT chess-square;

```

This restricts the value of the rank (column) and file (row) of a chess-square to be between 1 and 8.

4.3.4 Attributes

Most constructs in CML2 can have attributes. An attribute is a (possibly multi-valued) function into a value set. A number of value sets are assumed to be predefined, see type-range (page 14). The value sets can also be defined by the user, see value-type (page 14). The value set of an attribute cannot be another construct, relations between constructs have to be modelled as a (binary) relation.

```

attributes        ::= attributes : attribute+ .

attribute         ::= Attribute : type-range ;
                  [ cardinality ]
                  [ differentiation-of : Attribute ( Concept ) ; ]
                  [ default-value : Value ; ] .

```

Semantics. The cardinality of an attribute defines how many values that particular attribute may take. If the cardinality is omitted it is assumed to be precisely one. An attribute can be a differentiation of an attribute of a super construct, both the name and value set of the attribute can be differentiated. Consider the following example.

```

CONCEPT vehicle;

```

```

ATTRIBUTES:
  wheels: INTEGER;
END CONCEPT vehicle;

CONCEPT human;
  ATTRIBUTES:
    legs: INTEGER;
    DIFFERENTIATION-OF: wheels(vehicle);
END CONCEPT human;

```

4.3.5 Rule type

```

rule-type ::= rule-type Rule-type ;
           [ terminology ]
           rule-type-body
           [ examples : "Text" ; ]
           end rule-type [ Rule-type ; ] .

rule-type-body ::= constraint-rule-type | implication-rule-type .

constraint-rule-type ::= constraint : user-defined-type ;
                       [ cardinality ] .

implication-rule-type ::= antecedent : user-defined-type ;
                        [ cardinality ]
                        consequent : user-defined-type ;
                        [ cardinality ]
                        connection-symbol : Name ; .

```

4.3.6 Mathematical model

```

mathematical-model ::= mathematical-model Mathematical-model ;
                     [ terminology ]
                     [ parameters : parameter+ ]
                     [ equations : equation-list ]
                     end mathematical-model [ Mathematical-model ; ] .

parameter ::= Parameter : type-range ; .

equation-list ::= < equation | model-reference > + .

model-reference ::= model Mathematical-model ( [ function-arguments ] ) .

```

4.3.7 Structure

```

structure ::= structure Structure ;
           [ terminology ]
           [ sub-type-of : Structure , ... ; ]
           form : "Text " ;
           [ attributes ]
           [ axioms ]
           end structure [ Structure ; ] .

```

Structure. The notion of structure is used to describe objects with an internal structure that the knowledge engineer does not want to describe (at this moment) in detail. The *form* slot can be used to describe the structure.

4.3.8 Relation

The notion of relation is a central construct in modelling a domain. In CML2 we allow various forms of relations to cater for the specific requirements imposed by KBSs. The relation construct is used to link any type of objects to each other, including concepts, structures and relations. CML2 supports two types of relation arguments: a single concept; and a set of such objects.

```

relation ::= relation Relation ;
           [ terminology ]
           [ sub-type-of : Relation , ... ; ]
           arguments : argument+
           [ attributes ]
           [ axioms ]
           end relation [ Relation ; ] .

argument ::= argument-type ;
           [ role : Role ; ]
           [ cardinality ] .

argument-type ::= domain-construct-type |
                 set-of domain-construct-type |
                 list-of domain-construct-type .

domain-construct-type ::= built-in-type | user-defined-type .

built-in-type ::= object | concept | rule-type | structure |
                 relation | binary-relation |
                 mathematical-model | value-type .

user-defined-type ::= Concept | Rule-type | Structure |
                    Relation | Binary-relation | Mathematical-model .

```

4.3.9 Binary relation

binary-relation ::= **binary-relation** *Relation* ;
 [terminology]
 [**sub-type-of** : *Relation* , ... ;]
 [**inverse** : *Relation* ;]
argument-1 : argument
argument-2 : argument
 [relation-type]
 [attributes]
 [axioms]
end binary-relation [*Relation* ;] .

relation-type ::= **transitive** | **asymmetric** | **symmetric** |
irreflexive | **reflexive** | **antisymmetric** .

4.3.10 Type range

type-range ::= primitive-type | primitive-range |
Value-type | { *String-value* , ... } .

primitive-type ::= **number** | **integer** | **natural** | **real** | **image** |
string | **boolean** | **universal** | **date** | **text** .

primitive-range ::= **number-range** open-bracket *Number* , max-number close-bracket |
integer-range open-bracket *Integer* , max-integer close-bracket .

4.3.11 Value type

value-type ::= **value-type** *Value-type* ;
 [terminology]
 [**type** : **nominal** | **ordinal** ;]
 < **value-list** : { *Value* , ... } >
 | < **value-specification** : primitive-type | "Text" > ;
 [attributes]
end value-type [*Value-type* ;] .

4.4 Knowledge base

```

knowledge-base      ::= knowledge-base Knowledge-base ;
                       [ terminology ]
                       use : knowledge-base-use , ... ;
                       [ [ instances : ] ⟨ instance | tuple ⟩ + ]
                       [ variables : variable-declaration ; ... ; ]
                       [ expressions : knowledge-base-expression ... ; ]
                       [ annotations : "Text " ; ]
                       [ attributes ]
                       end knowledge-base [ Knowledge-base ; ] .

knowledge-base-use  ::= Domain-schema | Rule-type from Domain-schema .

variable-declaration ::= Variable , ... : Variable-type ; .

knowledge-base-expression ::= variable-declaration |

                           rule-type-instance |
                           "Text " .

rule-type-expression ::= equation |
                       type-operator rule-type-expression |
                       rule-type-expression part-operator rule-type-expression .

type-operator       ::= sub-type-of | super-type-of | type-of .

part-operator       ::= has-part | dimension | role .

```

4.4.1 Instances

A knowledge base normally contains instances of the constructs defined in a domain schema. The constructs for which instances can be defined is listed in the following table.

Domain schema	Knowledge base	Defines
concept	instance	Attribute values and parts
binary-relation	tuple	Arguments and attribute values
relation	tuple	Arguments and attribute values

Instances of concepts have a name to uniquely identify them. The names of these instances can then be referred to in instances of relations. It is not necessary for names of instance to be meaningful at all, they can be arbitrary identifiers.

Example. A simple domain schema for players who can participate in tournaments is:

```

DOMAIN-SCHEMA tournament-participation;

CONCEPT player;
  ATTRIBUTES:
    name: STRING;

```

```
        nationality: STRING;
END CONCEPT player;

CONCEPT tournament;
  ATTRIBUTES:
    city: STRING;
    participants: INTEGER;
    dates: STRING;
    rounds: INTEGER;
END CONCEPT tournament;

BINARY-RELATION played-in;
  ARGUMENT-1:
    player;
  ARGUMENT-2:
    tournament;
  ATTRIBUTES:
    score: REAL;
END BINARY-RELATION played-in;

END DOMAIN-SCHEMA tournament-participation;

For a particular tournament, for example the first match for the chess World Championship, we then
have these instances:

KNOWLEDGE-BASE 'World Chess Championships';
  USE: tournament-participation;

  INSTANCE steinitz;
    INSTANCE-OF: player;
    ATTRIBUTES:
      name: 'William Steinitz';
      nationality: AUT;
  END INSTANCE

  INSTANCE zukertort;
    INSTANCE-OF: player;
    ATTRIBUTES:
      name: 'Johannes Zukertort';
      nationality: POL;
  END INSTANCE

  INSTANCE 'WCC 01';
    INSTANCE-OF: tournament;
    ATTRIBUTES:
      city: 'New York, St Louis, New Orleans (USA)';
      dates: 'January 3 - March 11, 1886';
```

```

    participants: 2;
    rounds: 20;
END INSTANCE

TUPLE
  INSTANCE-OF: played-in;
  ARGUMENT-1: steinitz;
  ARGUMENT-2: 'WCC 01';
  ATTRIBUTES:
    score: 12.5;
END TUPLE

TUPLE
  INSTANCE-OF: played-in;
  ARGUMENT-1: zukertort;
  ARGUMENT-2: 'WCC 01';
  ATTRIBUTES:
    score: 7.5;
END TUPLE
END KNOWLEDGE-BASE 'World Chess Championships';

```

The example illustrates that for each instance, obviously, the construct that defines the instances must be given (with **instance-of**). The arguments of a tuple refer to the *names* of instances.

```

instance ::= instance Instance ;
           [ terminology ]
           instance-of : user-defined-type ;
           [ has-parts :
             < Instance ; [ role : Role ; ] > + ]
           [ attributes :
             < Attribute : Value ; > + ]
           end instance [ Instance ; ] .

tuple ::= tuple
         [ terminology ]
         instance-of : user-defined-type ;
         [ < argument-1 : Instance ;
           argument-2 : Instance ; > |
           < arguments : Instance , ... ; > ]
         [ attributes :
           < Attribute : Value ; > + ]
         end tuple .

```

4.5 Inference knowledge

inference-knowledge ::= **inference-knowledge** *Inference-knowledge* ;
 [terminology]
 [**use** : use-construct , ... ;]
 ⟨ inference |
 knowledge-role |
 transfer-function ⟩ *
end inference-knowledge [*Inference-knowledge* ;] .

4.5.1 Inference

inference ::= **inference** *Inference* ;
 [terminology]
 [**operation-type** : *Name* ;]
roles :
 input : *Dynamic-knowledge-role* , ... ;
 output : *Dynamic-knowledge-role* , ... ;
 [**static** : *Static-knowledge-role* , ... ;]
 [specification]
end inference [*Inference* ;] .

4.5.2 Transfer function

transfer-function ::= **transfer-function** *Transfer-function* ;
 [terminology]
type : ⟨ **provide** | **receive** | **obtain** | **present** ⟩
roles :
 input : *Dynamic-knowledge-role* , ... ;
 output : *Dynamic-knowledge-role* , ... ;
end transfer-function *Transfer-function* ; .

4.5.3 Knowledge role

knowledge-role ::= **knowledge-role** *Knowledge-role* ;
 [terminology]
type : **static** | **dynamic** ;
domain-mapping :
 ⟨ dynamic-domain-reference |
 static-domain-reference ⟩ ;
end knowledge-role *Knowledge-role* ; .

dynamic-domain-reference ::= domain-construct-type |
set-of domain-construct-type |
list-of domain-construct-type .

static-domain-reference ::= domain-construct-type **from** *Knowledge-base* .

4.6 Task knowledge

```

task-knowledge ::= task-knowledge Task-knowledge ;
                [ terminology ]
                [ use : Inference-knowledge , ... ; ]
                task-element*
end task-knowledge [ Task-knowledge ; ] .

```

4.6.1 Task

```

task ::= task Task
       [ terminology ]
       [ domain-name : Domain ; ]
       [ goal : "Text " ; ]
       roles :
         input : role-description+
         output : role-description+
       [ specification : "Text " ; ]
end task [ Task ; ] .

```

```

role-description ::= Task-role : "Text " ; .

```

4.6.2 Task method

The decomposition of a task method allows the specification of a function if it is not known whether the decomposition is an inference or a tasks. This facilitates the construction of flexible libraries.

```

task-method ::= task-method Task-method ;
              [ realizes : Task ; ]
              task-decomposition
              [ roles : intermediate : role-description+ ]
              control-structure : control-structure
              [ assumptions : "Text " ; ]
end task-method [ Task-method ; ] .

```

4.7 Control structure

```

control-structure ::= pseudo-code .
pseudo-code ::= statement+ | { { pseudo-code } } .
statement ::= function-call ; |
            control-loop |
            conditional-statement |
            role-operation |
            "Text " ; .
function-call ::= function ( [ proc-input ] [ ' ->' proc-output ] ) ; .

```

function	::= <i>Task</i> <i>Inference</i> transfer-function .
proc-input	::= <i>Role</i> `+'
proc-output	::= <i>Role</i> `+'
control-loop	::= ⟨ repeat pseudo-code until control-condition ; end repeat ⟩ ⟨ while control-condition do pseudo-code end while ⟩ ⟨ for-each <i>Role</i> in <i>Role</i> do pseudo-code end for-each ⟩ .
control-condition	::= ⟨ has-solution function-call ⟩ ⟨ new-solution function-call ⟩ ⟨ empty <i>Role</i> ⟩ ⟨ control-condition and control-condition ⟩ ⟨ control-condition or control-condition ⟩ ⟨ control-condition xor control-condition ⟩ ⟨ not control-condition ⟩ ⟨ size <i>Role</i> comparison-operator <i>Integer</i> ⟩ ⟨ <i>Role</i> comparison-operator <i>Value</i> ⟩ ⟨ (control-condition) ⟩ "Text " .
role-operation	::= <i>Role</i> `:= ' role-expression ; .
role-expression	::= <i>Role</i> ⟨ unary-role-operator role-expression ⟩ ⟨ role-expression binary-role-operator role-expression ⟩ .
binary-role-operator	::= add delete subtract .
unary-role-operator	::= member select select-random .

4.8 PSM knowledge

psm-knowledge	::= psm-knowledge <i>Psm-knowledge</i> [terminology] psm-description* end psm-knowledge [<i>Psm-knowledge</i> ;] .
---------------	---

4.8.1 PSM

```

psm ::= psm Psm ;
      [ terminology ]
      [ can-realize : problem-type , ... ; ]
      decomposition :
        functions : Function , ... ;
      roles :
        input : role-description+
        output : role-description+
        intermediate : role-description+
      control-structure : control-structure
      [ competence : "Text " ; ]
      [ assumptions : "Text " ; ]
      [ pragmatic-concerns : "Text " ; ]
      [ cost : "Text " ; ]
      [ utility : "Text " ; ]
      [ communication-protocol : "Text " ; ]
      end psm Psm ; .

problem-type ::= assessment | assignment | classification |
                 configuration | design | diagnosis |
                 modelling | monitoring | planning |
                 prediction | scheduling | "Text " .

```

4.9 Ontology mapping

Note. An elaborate description of ontology mappings is the subject of further research.

```

ontology-mapping ::= ontology-mapping Ontology-mapping ;
                   [ terminology ]
                   from : Domain-schema ;
                   to : Domain-schema ;
                   mappings : "Text " ;
                   end ontology-mapping [ Ontology-mapping ; ] .

```

4.10 Equations

The equation syntax is adopted from NMF (Neutral Model Format). NMF is an emerging standard for the definition of mathematical models. The basic entry point is equation (page 22).

A description of the operators and their precedence is given in operator-precedence (page 6).

4.10.1 Equation

equation	::=	' (' equation ') ' sign-operator equation negation-operator equation equation arithmetic-operator equation equation logical-operator equation equation comparison-operator equation equation dereference-operator equation equation equation-operator equation unsigned-constant variable-expression function-expression conditional-expression .
variable-expression	::=	<i>Variable</i> [derivative] [subscripts] .
subscripts	::=	' [' equation , ... '] ' .
unsigned-constant	::=	<i>Unsigned-integer</i> <i>Unsigned-real</i> "Text" .
function-expression	::=	<i>Function</i> (equation , ...) .
conditional-expression	::=	if equation then equation [else equation] end if .

4.10.2 Operators

equivalence-operator	::=	' = ' .
assignment-operator	::=	' := ' .
sign-operator	::=	' + ' ' - ' .
negation-operator	::=	not .
arithmetic-operator	::=	' + ' ' - ' ' * ' ' / ' ' ** ' .
logical-operator	::=	and or xor .
implication-operator	::=	' -> ' ' <- ' ' <-> ' .
dereference-operator	::=	' . ' .
comparison-operator	::=	' < ' ' > ' ' <= ' ' >= ' ' == ' ' != ' .
derivative-operator	::=	' ' ' .

4.11 Support

4.11.1 Cardinality

cardinality	::=	cardinality : cardinality-spec ; .
-------------	-----	---

```
cardinality-spec ::= any |
                  Natural |
                  Natural "+" |
                  Natural "-" Natural .
```

4.11.2 Role

```
role ::= role : Role ; .
```

4.11.3 Terminology

```
terminology ::= [ description : "Text" ; ]
               [ sources : "Text" ; ]
               [ synonyms : Name , ... ; ]
               [ translation : Name , ... ; ] .
```

A construct can be annotated with a textual description and sources (textbook, dictionary) as well as with a list of synonyms and translations.

5 Parser

The CML parser reads a file written in CML2, checks the syntax for correctness, and optionally generates output in a variety of formats. The parser is based on the syntax of CML2 as described in section 4.

The main purpose of the CML parser is to support the user of CML2. This is necessary for the following reasons:

Checking syntactical correctness.

CML2 is syntactically a rich, and perhaps even a complicated, language. A parser is absolutely necessary to check the syntax for correctness. CML2 is also a semantically a rich language, the parser also checks for static-semantics, for example whether referenced constructs are defined.

Pretty-printing and document generation.

The readability of CML2 can be increased by formatting and pretty-printing. The CML parser has an option to turn the ASCII input into ECO which KADS22 can turn into HTML, \LaTeX , hypertext and graphical representations.

Interactive editing.

When CML2 needs to be made operational or when the user wants to edit CML2 interactively, existing files still need to be parsed. The parser can be used as a stand-alone program, but also as a client to interactive environments such as KADS22.

5.1 Disclaimer

The current version of the CML parser is provided on an as-is basis. The expected status is as follows. The parser should accept all correct CML2 files, i.e. the parser itself is thought to be correct. The

parser will generate correct ECO for use with the corresponding version of KADS22.

5.2 Syntactic units

5.2.1 Names of keywords

The names of keywords (built-in CML2 names) must be written in uppercase. Optionally, these keywords may be enclosed in single quotes. For example, the following refer to the keyword concept: `CONCEPT` and `'CONCEPT'`.

Either a hyphen or an underscore may be used as a delimiter within a keyword, for example `KNOWLEDGE-MODEL` and `KNOWLEDGE_MODEL` are both correct. Note that, for a full definition, there is a space between the keyword `END` and the type defined: `END CONCEPT`.

The use of uppercase for keywords may seem old-fashioned. The motivation for this is that part of CML2 can be used as a terminology-definition language. Standard elements of the language (e.g. the word “concept”) are often defined, for example:

```
CONCEPT concept;
...
END CONCEPT concept;

CONCEPT attribute;
...
END CONCEPT attribute;

BINARY-RELATION concept-attribute;
  ARGUMENT-1: concept;
  ARGUMENT-2: attribute;
  ATTRIBUTES:
    required: BOOLEAN;
  ...
END BINARY-RELATION concept-attribute;
```

5.2.2 Completeness

The CML parser does not require that the input contains a complete knowledge model. Any sequence of constructs that has a corresponding `END` is acceptable as input to the parser.

5.2.3 Names of constructs

The name of a user-defined construct can consist of letters, digits, hyphens and underbars. When other characters are necessary, the name must be enclosed in single quotes, for example: `'Monster of Loch Ness'`. Users should take into account that, when generating output, the use of special characters may not be acceptable to the target formalism (e.g. C++ will not accept hyphens in names).

To alleviate this problem, the parser has a number of options to control the generation of special characters (see `-nohyphen`, `-nounderbar`, `-nospace`, `-nolower` and `-noupper`).

5.2.4 Unresolved constructs

The CML2 syntax and the CML parser make a distinction between a definition of a construct and a reference to a construct. The following may serve as an example of this distinction:

```
CONCEPT human;  
  SUPER-TYPE-OF: man, woman;  
  COMPLETE: YES;  
  DISJOINT: YES;  
END CONCEPT car;
```

In this example, the concept *human* is defined and the concepts *man* and *woman* are referred to.

A construct is *unresolved* when it is referred to but not defined. The CML parser accepts unresolved constructs, but can obviously not generate correct output in all cases.

The `-unresolved` option can be used as an aid to detect unresolved definitions and possible misspellings. The above example displays the following output:

```
% cml2 -unresolved human.cml  
  
File correctly parsed  
  
Unresolved names:  
man (guess: CONCEPT)  
woman (guess: CONCEPT)
```

5.3 Options

Below are the options for the CML parser. All options start with a hyphen and are optionally followed by arguments.

A list of options and their (default) value can be obtained by:

```
% cml2 ... -options
```

`-banner`

Prints a banner with the version, author information and the status. The default is not to print a banner.

`-eco`

This option generates output suitable for use by programs which adhere to the ECO notation. The default output file is `output.eco`.

`-embed start end`

The CML parser normally assumes that the entire input is correct CML2. It often happens that a document contains both CML2 and other material and you wish to check that the CML2 is correct without keeping it in separate files. A typical example would be a \LaTeX document:

Here is a description of my concept:

```
\begin{cml}
CONCEPT my-concept;
  ATTRIBUTES: my-attribute: UNIVERSAL;
END CONCEPT my-concept;
\end{cml}
```

More text follows...

With the `-embed` option you can specify how the CML parser can recognise CML2 in the input file. The first argument is a string that initiates CML2 and the second argument is a string that terminates CML2. In the above example the arguments would be (note that special characters need to be quoted in the shell):

```
-embed "\begin{cml}" "\end{cml}"
```

`-help`

Provides a summary of the most important options.

`-log`

Creates a log-file output `.log` with error messages and status information. Turns on `-silent`.

`-nohyphen`

Means that hyphens (“-”) are not acceptable in the target formalism. If underbars are acceptable (see `-nounderbar`), then all hyphens are replaced by underbars, otherwise hyphens are removed. By default, hyphens are acceptable.

`-nolower`

Means that lowercase characters are not acceptable in the target formalism. All such characters are replaced by their uppercase equivalent. By default, the target formalism is considered to be case insensitive.

`-nospace`

Means that spaces (and all other special characters) are not acceptable in the target formalism. If hyphens are acceptable (`-nohyphen`), then all spaces are replaced by hyphens, if underbars are acceptable (`-nounderbar`) then spaces are replaced by underbars, otherwise spaces are removed. By default, spaces and special characters are acceptable.

`-notes`

Provides a brief overview of recent changes to the parser or CML2.

-nounderbar

Means that underbars (“_”) are not acceptable in the target formalism. If hyphens are acceptable (`-nohyphen`), then all underbars are replaced by hyphens, otherwise underbars are removed. By default, hyphens are acceptable.

-noupper

Means that uppercase characters are not acceptable in the target formalism. All such characters are replaced by their lowercase equivalent. By default, the target formalism is considered to be case insensitive.

-o *basename*

Normally the output is stored in a file called `output.*`. This option changes the output file to `basename.*`. The extension of the output file name depends on the generation option, see `-eco`.

-options

When this option is encountered in the argument list, the values of all options are printed on the console.

-silent

Only explicitly requested warnings and messages are shown on the console. Normally, the parser also outputs semi-redundant information (for example to say that the parser has started). Such messages are suppressed by this option. This option is off by default.

-st

Prints the *symbol table*.

-trace

For development purposes only.

-unresolved

Lists the names that are unresolved.

-verbose

Prints information about progress on the console. Most of the information is for development purposes only. This option is off by default.

-version

Prints the version number and last change date.

5.4 Examples

Typical usage is:

```
% cml -eco vt.cml
```

Parses the file `vt.cml` and generates ECO output in `output.eco`.

```
% cml -o vt -eco vt.cml
```

Parses the file `vt.cml` and generates ECO output in `vt.eco`.

5.5 Error recovery

The CML parser attempts to recover from errors in the input. Error recovery is very rudimentary: all input until the next semi-colon is skipped before parsing continues. This may result in error messages that are the consequence of an earlier error rather than a genuine error.

5.6 Error messages

Error messages are printed on the console. The following error messages are the most frequent. An error message is normally preceded by a number, this is the approximate line number of the error.

The format of error messages is compatible with the `M-x compile` and `C-`` commands of Emacs.

Semi-colon expected.

The parser expected a semi-colon, but it was missing. In most cases the parser can insert the semi-colon and continue parsing. This is a warning.

Colon expected.

The parser expected a colon, but it was missing. In most cases the parser can insert the colon and continue parsing. This is a warning.

Referred type “name” not defined.

This message indicates that a user-defined type is implied in the input, but is not defined anywhere (a user-defined type is a concept, relation, etc.). Check the input to see where “name” is defined and either correct the misspelling or insert a definition for the given type.

It is possible to leave the name undefined and import it from another ontology with the CML2 USE construct. The parser cannot resolve such imports and will still give a warning, other CML2 tools (such as VOID) will resolve the name and the warning can be ignored.

Redefinition of “name”.

This message indicates that a user-defined type is defined more than once. This frequently happens with relation names (“consists-of” is a favourite). Ensure that all user-defined types are defined precisely once.

Index

- add (keyword), 20
- and (keyword), 20, 22
- annotations (keyword), 15
- antecedent (keyword), 12
- antisymmetric (keyword), 14
- any (keyword), 23
- argument (non-terminal), 13
- argument-1 (keyword), 14, 17
- argument-2 (keyword), 14, 17
- argument-type (non-terminal), 13
- arguments (keyword), 13, 17
- arithmetic-operator (non-terminal), 22
- assessment (keyword), 21
- assignment (keyword), 21
- assignment-operator (non-terminal), 22
- assumptions (keyword), 19, 21
- asymmetric (keyword), 14
- attribute (non-terminal), 11
- attributes (keyword), 11, 17
- attributes (non-terminal), 11
- axioms (keyword), 11
- axioms (non-terminal), 11

- binary-relation (keyword), 13, 14
- binary-relation (non-terminal), 14
- binary-role-operator (non-terminal), 20
- boolean (keyword), 14
- built-in-type (non-terminal), 13

- can-realize (keyword), 21
- cardinality (keyword), 22
- cardinality (non-terminal), 22
- cardinality-spec (non-terminal), 23
- class, 10
- classification (keyword), 21
- communication-protocol (keyword), 21
- comparison-operator (non-terminal), 22
- competence (keyword), 21
- complete (keyword), 10, 11
- concept (keyword), 10, 13
- concept (non-terminal), 10
- conditional-expression (non-terminal), 22
- configuration (keyword), 21
- connection-symbol (keyword), 12

- consequent (keyword), 12
- constraint (keyword), 12
- constraint-rule-type (non-terminal), 12
- control-condition (non-terminal), 20
- control-loop (non-terminal), 20
- control-structure (keyword), 19, 21
- control-structure (non-terminal), 19
- cost (keyword), 21

- date (keyword), 14
- decomposition (keyword), 21
- default-value (keyword), 11
- definitions (keyword), 10
- delete (keyword), 20
- dereference-operator (non-terminal), 22
- derivative-operator (non-terminal), 22
- description (keyword), 23
- design (keyword), 21
- diagnosis (keyword), 21
- differentiation-of (keyword), 11
- dimension (keyword), 11, 15
- disjoint (keyword), 10, 11
- do (keyword), 20
- domain-construct (non-terminal), 10
- domain-construct-type (non-terminal), 13
- domain-knowledge (keyword), 10
- domain-knowledge (non-terminal), 10
- domain-mapping (keyword), 18
- domain-name (keyword), 19
- domain-schema (keyword), 10
- domain-schema (non-terminal), 10
- dynamic (keyword), 18
- dynamic-domain-reference (non-terminal), 18

- else (keyword), 22
- empty (keyword), 20
- end (keyword), 9, 10, 12–15, 17–22
- entity, 10
- equation (non-terminal), 22
- equation-list (non-terminal), 12
- equations (keyword), 12
- equivalence-operator (non-terminal), 22
- examples (keyword), 12
- expertise model, 9

- expressions (keyword), 15
- for-each (keyword), 20
- form (keyword), 13
- from (keyword), 10, 15, 18, 21
- function (non-terminal), 20
- function-call (non-terminal), 19
- function-expression (non-terminal), 22
- functions (keyword), 21
- goal (keyword), 19
- has-part (keyword), 15
- has-part (non-terminal), 11
- has-parts (keyword), 10, 17
- has-solution (keyword), 20
- if (keyword), 22
- image (keyword), 14
- implication-operator (non-terminal), 22
- implication-rule-type (non-terminal), 12
- in (keyword), 20
- inference (keyword), 18
- inference (non-terminal), 18
- inference-knowledge (keyword), 18
- inference-knowledge (non-terminal), 18
- input (keyword), 18, 19, 21
- instance (keyword), 17
- instance (non-terminal), 17
- instance-of (keyword), 17
- instances (keyword), 15
- integer (keyword), 14
- integer-range (keyword), 14
- intermediate (keyword), 19, 21
- inverse (keyword), 14
- irreflexive (keyword), 14
- knowledge-base (keyword), 15
- knowledge-base (non-terminal), 15
- knowledge-base-expression (non-terminal), 15
- knowledge-base-use (non-terminal), 15
- knowledge-model (keyword), 9
- knowledge-model (non-terminal), 9
- knowledge-role (keyword), 18
- knowledge-role (non-terminal), 18
- list-of (keyword), 13, 18
- logical-operator (non-terminal), 22
- mappings (keyword), 21
- mathematical-model (keyword), 12, 13
- mathematical-model (non-terminal), 12
- member (keyword), 20
- model (keyword), 12
- model-reference (non-terminal), 12
- modelling (keyword), 21
- monitoring (keyword), 21
- natural (keyword), 14
- negation-operator (non-terminal), 22
- new-solution (keyword), 20
- no (keyword), 10, 11
- nominal (keyword), 14
- not (keyword), 20, 22
- number (keyword), 14
- number-range (keyword), 14
- object (keyword), 13
- obtain (keyword), 18
- ontology-mapping (keyword), 21
- ontology-mapping (non-terminal), 21
- operation-type (keyword), 18
- or (keyword), 20, 22
- ordinal (keyword), 14
- output (keyword), 18, 19, 21
- parameter (non-terminal), 12
- parameters (keyword), 12
- part-of (keyword), 10
- part-operator (non-terminal), 15
- planning (keyword), 21
- pragmatic-concerns (keyword), 21
- prediction (keyword), 21
- present (keyword), 18
- primitive-range (non-terminal), 14
- primitive-type (non-terminal), 14
- problem solving method, 21
- problem-type (non-terminal), 21
- proc-input (non-terminal), 20
- proc-output (non-terminal), 20
- provide (keyword), 18
- pseudo-code (non-terminal), 19
- psm (keyword), 21
- psm (non-terminal), 21
- psm-knowledge (keyword), 20
- psm-knowledge (non-terminal), 20

- real (keyword), 14
- realizes (keyword), 19
- receive (keyword), 18
- reflexive (keyword), 14
- relation (keyword), 13
- relation (non-terminal), 13
- relation-type (non-terminal), 14
- repeat (keyword), 20
- role (keyword), 13, 15, 17, 23
- role (non-terminal), 23
- role-description (non-terminal), 19
- role-expression (non-terminal), 20
- role-operation (non-terminal), 20
- roles (keyword), 18, 19, 21
- rule-type (keyword), 12, 13
- rule-type (non-terminal), 12
- rule-type-body (non-terminal), 12
- rule-type-expression (non-terminal), 15

- scheduling (keyword), 21
- select (keyword), 20
- select-random (keyword), 20
- set-of (keyword), 13, 18
- sign-operator (non-terminal), 22
- size (keyword), 20
- sources (keyword), 23
- specification (keyword), 19
- statement (non-terminal), 19
- static (keyword), 18
- static-domain-reference (non-terminal), 18
- string (keyword), 14
- structure (keyword), 13
- structure (non-terminal), 13
- sub-type-of (keyword), 10, 13–15
- subscripts (non-terminal), 22
- subtract (keyword), 20
- super-type-of (keyword), 10, 15
- symmetric (keyword), 14
- synonyms (keyword), 23

- task (keyword), 19
- task (non-terminal), 19
- task-knowledge (keyword), 19
- task-knowledge (non-terminal), 19
- task-method (keyword), 19
- task-method (non-terminal), 19
- terminology (non-terminal), 23

- text (keyword), 14
- then (keyword), 22
- to (keyword), 21
- transfer-function (keyword), 18
- transfer-function (non-terminal), 18
- transitive (keyword), 14
- translation (keyword), 23
- tuple (keyword), 17
- tuple (non-terminal), 17
- type (keyword), 14, 18
- type-of (keyword), 15
- type-operator (non-terminal), 15
- type-range (non-terminal), 14

- unary-role-operator (non-terminal), 20
- universal (keyword), 14
- unsigned-constant (non-terminal), 22
- until (keyword), 20
- use (keyword), 10, 15, 18, 19
- use-construct (non-terminal), 10
- user-defined-type (non-terminal), 13
- utility (keyword), 21

- value-list (keyword), 14
- value-specification (keyword), 14
- value-type (keyword), 13, 14
- value-type (non-terminal), 14
- variable-declaration (non-terminal), 15
- variable-expression (non-terminal), 22
- variables (keyword), 15
- viewpoint (non-terminal), 11
- viewpoints (keyword), 10

- while (keyword), 20

- xor (keyword), 20, 22

- yes (keyword), 10, 11