BRICS NS-96-15 CoFI: CASL – The CoFI Algebraic Specification Language

BRICS

Basic Research in Computer Science

CASL
– The CoFI Algebraic Specification Language

Tentative Design: Language Summary

CoFI

BRICS Notes Series NS-96-15

ISSN 0909-3206 December 1996

See back inner page for a list of recent publications in the BRICS Notes Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through World Wide Web and anonymous FTP:

`http://www.brics.dk/`
`ftp://ftp.brics.dk/`
This document in subdirectory `NS/96/15/`

CASL – The CoFI Algebraic Specification Language
Tentative Design: Language Summary

CoFI: The Common Framework Initiative for Algebraic Specification*

Version 0.95 (20 December 1996)

This document is available for browsing on WWW[1]

*CoFI is an international collaborative effort, coordinated by Peter D. Mosses. Home Page: http://www.brics.dk/Projects/CoFI/

[1] http://www.brics.dk/Projects/CoFI/Documents/Tentative/LanguageSummary/

Abstract

This is the official summary of the Tentative Design of CASL, the CoFI Algebraic Specification Language. Please take a close look at it, and report any mistakes or inconsistencies that you notice, as well as pointing out any parts that still need further clarification (bearing in mind the informal nature of this summary). All comments on the Tentative Design of CASL and on this summary should be sent to the CoFI Language Design mailing list[2] .

This document will be the basis for closer investigation of the tentative language design by various CoFI task groups (Semantics, Tools, Methodology, Language Design), leading to a firm CASL Design Proposal in March 1997.

The language CASL summarized here is central to CoFI: it is a reasonably expressive algebraic language for specifying requirements and design for conventional software. From CASL, simpler CoFI languages (e.g., for interfacing with existing tools) are to be obtained by restriction, and CASL is to be incorporated in more advanced CoFI languages (e.g., for specifying reactive systems). CASL strikes a balance between simplicity and expressiveness. The main features of its design are as follows:

Many-sorted *basic specifications* in CASL denote classes of many-sorted partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed. Axioms are first-order formulae built from definedness assertions and both strong and existential equations. Sort generation constraints can be stated. Type definitions are provided for concise specification of enumerations and products. Subsorted basic specifications provide moreover a simple treatment of subsorts, interpreting them as embeddings.

Structured specifications allow translation, reduction, union, and extension of specifications. Extensions may be required to be persistent and/or free; initiality constraints are a special case. A simple form of generic (parametrized) specifications is provided, together with instantiation involving parameter-fitting translations.

Architectural specifications express that the specified software is to be composed from separately-developed, reusable units with clear interfaces.

Finally, *libraries* allow the distributed storage and retrieval of named specifications.

For further details of the original requirements and functionality for the CoFI Common Language for Algebraic Specification, see the Working Documents “Meta-Requirements” and “Meta-Design”, accessible via the CoFI Home Page.

[2] mailto:cofi-language@brics.dk

Contents

About this document

This summary of the tentative design of **CASL** is intended for readers who are already familiar with the main concepts involved, which are therefore reviewed only briefly.

The **CoFI** Semantics, Tools, Methodology, and Reactive Systems task groups are to assess the tentative design from their various perspectives, at the same time starting to prepare some of the relevant associated documents that should eventually accompany the **CASL** Reference Manual. A Higher-Order Task Group is to be set up, to investigate extensions of **CASL** to include higher-order functions. The Language Design Task Group is to design a concrete syntax for **CASL** (the tentative language design provides only the abstract syntax) in collaboration with the Tools Task Group, and to prepare examples of specifications.

Provided that the changes to the tentative design subsequently suggested by the various task groups are not too dramatic, the plan is to converge on a firm **CASL** Design Proposal by March 1997. This proposal should then be open to public debate for at least a month, before submission to the sponsoring IFIP Working Group on Foundations of System Specification (WG 1.3) for approval, in May 1997.

Note, however, that the Common Framework Initiative is currently a voluntary, unfunded effort, and the plans outlined above may change, in response to fluctuations in the available human resources.

Contributors

The **CoFI** Language Design Task Group was formed at the founding meeting of the Common Framework Initiative, in Oslo, September 1995. The working meetings held in Paris (November 1995), Munich (January 1996), Oxford (March 1996), and Paris (May 1996) resulted in various language design study notes, which helped to guide the subsequent design of the **CoFI** Algebraic Common Language, **CASL**.

All participants at the major Language Design Task Group meeting in Munich (July 1996) have contributed in one way or another to the tentative design of **CASL**; they were: Egidio Astesiano, Didier Bert, Jan Bergstra, Gilles Bernot, Michel Bidoit, Maura Cerioli, Christine Choppy, Radu Grosu, Anne Haxthausen, Bernd Krieg-Brückner (chairman), Peter D. Mosses, Fernando Orejas, Olaf Owe, Don Sannella, Andrzej Tarlecki, Michał Walicki, and Martin Wirsing.

This document has been developed by Peter D. Mosses, originally on the basis of the design proposals and notes made available before the Munich meeting and the agreements reached during the meeting. Its preparation was greatly assisted by the timely production of the minutes of the Munich meeting by Christine Choppy. Moreover, the Language Design Task Group participants who attended the Dagstuhl seminar (July 1996), namely Egidio Astesiano, Michel Bidoit, Maura Cerioli, Peter D. Mosses, Don Sannella, Andrzej Tarlecki, and Martin Wirsing, helped reach essential clarifications on several outstanding issues.

Subsequent versions of this document have attempted to incorporate the improvements suggested in various comments. Comments on versions 0.9 and 0.91 of this document were received from: Didier Bert, Maura Cerioli, Pietro Cenciarelli, Bernd Krieg-Brückner, Erik Saaman, and Andrzej Tarlecki. Comments on version 0.92 were received from: Anne Haxthausen, Bernd Krieg-Brückner, Till Mossakowski, Don Sannella, Andrzej Tarlecki, and Michał Walicki. Comments on version 0.93 were received from: Didier Bert, Michel Bidoit, Maura Cerioli, Christine Choppy, Chris George, Radu Grosu, Jim Horning, Bernd Krieg-Brückner, Till Mossakowski, Peter D. Mosses, Gianna Reggio, Don Sannella, Andrzej Tarlecki, Eelco Visser, and Michał Walicki. The proposed treatment of subsorts is based on a study note[3] by Maura Cerioli, Till Mossakowski, Anne Haxthausen, Bernd Krieg-Brückner, Peter D. Mosses, and Olaf Owe. The proposed treatment of generic specifications is based on a study note[4] by Michel Bidoit. The proposed treatment of architectural specifications is based on a study note[5] by Peter D. Mosses. The comments have been sent to the mailing list[6] subscribers, and are accessible in the mailing list archives[7].

A working meeting of the Language Design Task Group in Edinburgh (November 1996) helped to settle some remaining issues. The participants were: Egidio Astesiano, Hubert Baumeister, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner (chairman), Till Mossakowski, Peter D. Mosses, Don Sannella, Andrzej Tarlecki, and Michał Walicki. The participants at the Edinburgh meeting also decided to adopt the acronym **CASL** for the **CoFI** Algebraic Specification Language; it was originally proposed by Christine Choppy.

Final comments on the draft version of the present document were provided by: Michel Bidoit, Maura Cerioli, Hélène Kirchner, Bernd Krieg-Brückner, Till Mossakowski, Peter D. Mosses, Gianna Reggio, Don Sannella, Andrzej Tarlecki

[3] http://www.brics.dk/Projects/CoFI/StudyNotes/Lang/MC++-1.html
[4] http://www.brics.dk/Projects/CoFI/StudyNotes/Lang/MB-2.html
[5] http://www.brics.dk/Projects/CoFI/StudyNotes/Lang/PDM-2.html
[6] mailto:cofi-language@brics.dk
[7] http://www.brics.dk/Projects/CoFI/MailingLists/cofi-language/96/maillist.html

Structure

Part I (Sections 1–4) deals with *basic specifications*—first many-sorted, then subsorted; Part II (Sections 5–6) provides *structured specifications*; Part III (Sections 7–8) summarizes so-called *architectural specifications*, which, in contrast to the foregoing specifications, prescribe the separate development of composable, reusable implementation units; and finally Part IV (Sections 9–10) considers *libraries* of definitions. For each kind of specification, the main underlying semantic concepts are summarized, then the abstract syntax of the language constructs is given and its semantics is described informally.

Appendix A provides a complete grammar for the abstract syntax of the language. The sort and constructor symbols used in the abstract syntax are reasonably suggestive of their intended interpretation, so reading this grammar may be the easiest way to get a quick overview of the language.

N.B. *No* consideration has yet been given to *concrete* syntax in the Tentative Design! The constructor symbols of the abstract syntax are *not* intended for use as key-words, nor is the order of components of constructs necessarily that to be used in concrete syntax.

Appendix B provides references to related documents.

Appendix C lists some proposed changes (additions) to the tentative design. These were discussed at the Edinburgh meeting and proposed in the draft of the present document, but it was felt that some further investigation was needed before incorporating them. (This investigation had not been completed by the time this document was released.)

The (rudimentary) Index may facilitate locating the places where terminology is explained.

Part I

Basic Specifications

1 Basic Concepts

First, here is an brief reminder of how specification frameworks may be formalized in terms of so-called *institutions* (some category-theoretic details are omitted) and *proof systems*.

A *basic specification framework* may be characterized by:

- a class **Sig** of *signatures* Σ, each determining the symbols whose intended interpretation is to be specified, with *morphisms* between signatures;
- a class $\mathbf{Mod}(\Sigma)$ of *models*, with *homomorphisms* between them, for each signature Σ;
- a set $\mathbf{Sen}(\Sigma)$ of *sentences* (or *axioms*), for each signature Σ;
- a relation $\models$ of *satisfaction*, between models and sentences over the same signature; and
- a *proof system*, for inferring sentences from sets of sentences.

A *basic specification* consists of a signature Σ together with a set of sentences from $\mathbf{Sen}(\Sigma)$. The signature provided for a particular sentence in a specification is called its *local environment*. (It may be a restriction of the entire signature of the specification, e.g., determined by an order of *presentation* for the signature declarations and the sentences.)

The (loose) semantics of a basic specification is the class of those models in $\mathbf{Mod}(\Sigma)$ which satisfy all the specified sentences. A specification is said to be *consistent* when there are some models that satisfy all the sentences, and *inconsistent* when there are no such models. A sentence is a *consequence* of a basic specification if it is satisfied in all the models of the specification.

A *signature morphism* $\sigma : \Sigma \to \Sigma'$ determines a *translation* function $\mathbf{Sen}(\sigma)$ on sentences, mapping $\mathbf{Sen}(\Sigma)$ to $\mathbf{Sen}(\Sigma')$, and a *reduct* function $\mathbf{Mod}(\sigma)$ on models, mapping $\mathbf{Mod}(\Sigma')$ to $\mathbf{Mod}(\Sigma)$. Satisfaction is required to be preserved by translation: for all $S \in \mathbf{Sen}(\Sigma)$, $M' \in \mathbf{Mod}(\Sigma')$,

$$\mathbf{Mod}(\sigma)(M') \models S \iff M' \models \mathbf{Sen}(\sigma)(S).$$

The proof system is required to be sound, i.e., sentences inferred from a specification are always consequences; moreover, inference is to be preserved by translation.

Sentences of basic specifications may include *constraints* that restrict the class of models, e.g., to reachable ones.

The rest of this section indicates the underlying signatures, models, sentences, and proof system for many-sorted basic specifications of the **CASL** specification framework; the extra components concerned with subsorts are deferred to Section 3. The abstract syntax of the language constructs used for many-sorted basic specifications is described in Section 2, with subsorting constructs deferred to Section 4. Any well-formed basic specification determines a signature and a set of sentences, the models of which provide the semantics of the basic specification.

1.1 Signatures

A *many-sorted signature* $\Sigma = (S, TF, PF, P)$ consists of:

- a set S of *sorts*;
- sets $TF_{w,s}$, $PF_{w,s}$, of *total function* resp. *partial function symbols*, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each *function profile* (w, s) consisting of a sequence of *argument sorts* $w \in S^*$ and a *result sort* $s \in S$ (*constants* are treated as functions with no arguments);
- sets P_w of *predicate* symbols, for each *predicate profile* consisting of a sequence of argument sorts $w \in S^*$.

Note that symbols may be *overloaded*, occurring in more than one of the above sets. To ensure that there is no ambiguity in sentences at this level, however, symbols are always qualified by profiles when used (the language considered in Section 2 allows the omission of such qualifications when these are unambiguously determined by the context).

A *many-sorted signature morphism* $\sigma : (S, TF, PF, P) \to (S', TF', PF', P')$ consists of a mapping from S to S', and for each $w \in S^*, s \in S$, a mapping between the corresponding sets of function, resp. predicate symbols. A partial function symbol may be mapped also to a total function symbol, but not vice versa.

1.2 Models

For a many-sorted signature $\Sigma = (S, TF, PF, P)$ a *many-sorted model* $M \in$ $\mathbf{Mod}(\Sigma)$ is a *many-sorted first-order structure* consisting of a *many-sorted partial algebra*:

- a (possibly-empty) *carrier set* s^M for each sort $s \in S$ (let w^M denote the Cartesian product $s_1^M \times \cdots \times s_n^M$ when $w = s_1 \ldots s_n$),

- a *partial function* f^M from w^M to s^M for each function symbol $f \in TF_{w,s}$ or $f \in PF_{w,s}$, the function being required to be total in the former case,

together with:

- a *predicate* $p^M \subseteq w^M$ for each predicate symbol $p \in P_w$.

A (weak) *many-sorted homomorphism* h from M_1 to M_2, with $M_1, M_2 \in \mathbf{Mod}(S, TF, PF, P)$, consists of a function $h_s : s^{M_1} \to s^{M_2}$ for each $s \in S$ preserving not only the values of functions but also their definedness, as well as the truth of predicates.

1.3 Sentences

The *many-sorted terms* on a signature $\Sigma = (S, TF, PF, P)$ and a set of sorted variables X are built from:

- variables from X;
- applications of qualified function symbols in $TF \cup PF$ to argument terms of appropriate sorts.

We refer to such terms as *fully-qualified terms*, to avoid confusion with the terms of the language considered in Section 2, which allow the omission of qualifications and explicit sorts when these are unambiguously determined by the context.

An *explicitly-sorted term* is formed by attaching a sort to a fully-qualified term of that sort.[8]

For a many-sorted signature $\Sigma = (S, TF, PF, P)$ the *many-sorted sentences* in $\mathbf{Sen}(\Sigma)$ are the usual closed many-sorted first-order logic formulae, built using quantification (over sorted variables) and logical connectives from the following *atomic formulae*:

- applications of qualified predicate symbols $p \in P$ to argument terms of appropriate sorts;
- assertions about the definedness of explicitly-sorted terms;
- existential and strong equations between explicitly-sorted terms of the same sort.

The sentences $\mathbf{Sen}(\Sigma)$ also include *sort-generation constraints*. Let $\Sigma = (S, TF, PF, P)$. A sort-generation constraint consists of (S', F') with $S' \subseteq S$ and $F' \subseteq TF \cup PF$.

[8]The notion of explicitly-sorted term is redundant in the many-sorted framework, but becomes significant when subsorts are introduced in Section 4.

1.4 Satisfaction

The satisfaction of a sentence in a structure M is determined as usual by the holding of its atomic formulae w.r.t. assignments of values to all the variables that occur in them, the value assigned to variables of sort s being in s^M. The value of a term w.r.t. a variable assignment may be undefined, due to the application of a partial function during the evaluation of the term. Note, however, that the satisfaction of sentences is 2-valued.

The application of a predicate symbol p to a sequence of argument terms holds in M iff the values of all the terms are defined and give a tuple belonging to p^M. A definedness assertion concerning a term holds iff the value of the term is defined. An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined.

The value of an occurrence of a variable in a term is that provided by the given variable assignment. The value of the application of a function symbol f to a sequence of argument terms is defined only if the values of all the argument terms are defined and give a tuple in the domain of definedness of f^M, and then it is the associated result value.

A sort-generation constraint (S', F') is satisfied in a Σ-model M if the carriers of the sorts in S' are *generated* by the function symbols in F'. I.e., every element of each sort in S' is the value of a term built from just these symbols (possibly using variables of sorts not in S', with appropriate assignments of values to them).

1.5 Proof System

The usual axioms and inference rules of total many-sorted first-order logic have to be adjusted to take into account possible undefinedness of the values of terms, the definedness assertions and the two kinds of equations. Sort-generation constraints introduce extra inference rules, corresponding to (structural) induction principles, the consequences of which cannot be fully captured by first-order logic.

2 Basic Constructs

This section provides the abstract syntax and determines the intended interpretation of the constructs of *many-sorted* basic specifications; constructs concerned with subsorts are deferred to Section 4. For an introduction to the form of abstract syntax used to indicate constructs in this document, see

Appendix A, which also collects together the abstract syntax of the entire CASL specification language. In later sections, an elided (...) alternative in the grammar indicates that the enclosing production is extending a production given earlier for the same sort of construct.

```
BASIC-SPEC      ::=   basic-spec BASIC-ITEM*
```

A *well-formed* many-sorted basic specification BASIC-SPEC of the CASL language determines a basic specification of the underlying many-sorted institution, consisting of a signature and a set of sentences of the form described in Section 1. The models of this signature and set of sentences provide the *semantics* of the basic specification. Thus this section explains well-formedness of basic specifications, and the way that they determine the underlying signatures and sentences, but in general it does not directly explain the intended interpretation of constructs.

Each BASIC-ITEM (apart from a VAR-DECL) may determine part of a signature and/or some sentences. The parts of signatures determined by the basic items in a basic specification are required to be disjoint; in particular, it is not allowed to specify the same BASIC-ITEM more than once in the same list BASIC-ITEM*. The order of the basic items is significant: all symbols used in a BASIC-ITEM must be in the local environment provided by the preceding BASIC-ITEMs, i.e., linear visibility of symbols.

```
BASIC-ITEM      ::=   SIG-DECL | VAR-DECL | AXIOM| SORT-GEN
```

A signature declaration SIG-DECL determines part of a signature. A variable declaration VAR-DECL determines sorted variables that are implicitly universally-quantified in the axioms of the enclosing BASIC-SPEC (variables may also be declared locally, by explicit quantification in axioms). Note that variable declarations do not contribute to the signature of the specification in which they occur. An AXIOM determines a sentence, as does a sort-generation constraint SORT-GEN.

2.1 Signature Declarations

```
SIG-DECL        ::=   SORT-DECL | FUN-DECL | PRED-DECL
```

A sort declaration SORT-DECL determines one or more sorts. A function declaration FUN-DECL determines one or more function symbols of a particular profile; similarly for a predicate declaration PRED-DECL. Function and predicate symbols may be overloaded, being declared with several different profiles in the same basic specification. The declaration of several symbols together in a single DECL is equivalent to their separate declarations in a series of similar DECLs.

```
ID             ::=   SIMPLE-ID
SIMPLE-ID      -- structure insignificant
```

The internal structure of simple identifiers SIMPLE-ID, used to identify sorts, functions, and predicates, is insignificant (ID is extended with compound identifiers in Section 6.3). The same identifier may be used for both a function and a predicate in the same BASIC-SPEC, but not for a sort and for a function or predicate.

2.1.1 Sorts

```
SORT-DECL      ::=   sort-decl SORT+
SORT           ::=   ID
```

A sort declaration SORT-DECL determines that each of the sorts in the list SORT+ is in the signature.

2.1.2 Constants and Functions

```
FUN-DECL       ::=   fun-decl FUN-NAME+ FUN-TYPE
FUN-TYPE       ::=   fun-type TOTALITY SORT* SORT
TOTALITY       ::=   total | partial
FUN-NAME       ::=   ID
```

A constant is treated as a function with no arguments (thus its value may be undefined when it is declared to be partial). A function declaration FUN-DECL determines that each function name in the list FUN-NAME+ is in the signature, being in the set of partial or total function symbols, and with the specified argument sorts and result sort, according to the function type FUN-TYPE.

2.1.3 Predicates

```
PRED-DECL      ::=   pred-decl PRED-NAME+ PRED-TYPE
PRED-TYPE      ::=   pred-type SORT*
PRED-NAME      ::=   ID
```

A predicate declaration PRED-DECL determines that each predicate name in the list PRED-NAME+ is in the signature, with the specified argument sorts according to the predicate type PRED-TYPE.

2.2 Variable Declarations

```
VAR-DECL       ::=   var-decl VAR+ SORT
VAR            ::=   SIMPLE-ID
```

A variable declaration VAR-DECL determines sorted variables for use in axioms; it does not contribute to the signature. Such a declaration of a variable as a BASIC-ITEM abbreviates adding a universal quantification on that variable to all axioms in the enclosing BASIC-SPEC. The variables in the list VAR+ must be distinct, and distinct from the variables previously declared by other VAR-DECLs in the same list of basic items A simple identifier SIMPLE-ID for a variable may not be used simultaneously as the identifier of a sort, function, or predicate.

2.3 Axioms and Terms

```
AXIOM          ::=  FORMULA
```

An axiom is well-formed when it is constructed from well-formed atomic formulae (with respect to the local environment), and, moreover, all the variables that occur in it are declared in enclosing quantifications or in variable declarations of the enclosing basic specification. A well-formed axiom determines a sentence of the underlying basic specification.

```
FORMULA        ::=  QUANTIFICATION | CONJUNCTION | DISJUNCTION
               |    IMPLICATION | EQUIVALENCE | NEGATION | ATOM
```

A formula is constructed from atomic formulae of the form ATOM using quantification and the usual logical connectives.

2.3.1 Quantifiers

```
QUANTIFICATION ::=  quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER     ::=  forall | exists | exists-uniquely
```

A quantification with more than one variable declaration VAR-DECL abbreviates a nest of quantifications with the same quantifier. A quantification with a declaration declaring more than one variable abbreviates a nest of quantifications with the same quantifier and sort. The scope of a variable declaration in a QUANTIFICATION is the component FORMULA, and an inner declaration for a variable with the same identifier as in an outer declaration overrides the outer declaration.

2.3.2 Logical Connectives

```
CONJUNCTION    ::=  conjunction FORMULA+
DISJUNCTION    ::=  disjunction FORMULA+
IMPLICATION    ::=  implication FORMULA FORMULA
EQUIVALENCE    ::=  equivalence FORMULA FORMULA
NEGATION       ::=  negation FORMULA
```

These formulae determine the usual logical connectives on sub-formulae. Conjunction and disjunction are generalized to lists of formulae `FORMULA+`, and are only well-formed when the list contains at least two formulae.

2.3.3 Atomic Formulae and Terms

```
ATOM          ::=  TRUTH | PREDICATION | DEFINEDNESS | EQUATION
```

An *atomic formula* is well-formed (with respect to the current declarations) if it is well-sorted and expands to a unique atomic formula for constructing sentences. The notions of when an atomic formula or term is *well-sorted* and of its *expansion* are indicated below for the various constructs. Due to overloading of predicate and/or function symbols, a well-sorted atomic formula or term may have several expansions, preventing it from being well-formed. Qualifications on function and predicate symbols may be used to determine the intended expansion and make it well-formed; explicit sorts on arguments and/or results may also help to avoid unintended expansions.

```
TRUTH         ::=  true | false
```

The atomic formulae `true` and `false` are always well-sorted, and expand to reserved, pre-declared constant predicate symbols, such that `true` always holds and `false` never holds.

```
PREDICATION   ::=  predication PRED-SYMB TERM*
PRED-SYMB     ::=  pred-symb PRED-NAME PRED-TYPE?
```

An application of a predicate symbol `PRED-SYMB` is well-sorted when there is a declaration of `PRED-SYMB` (with the specified `PRED-TYPE`, if any) such that all the argument terms are well-sorted for the declared argument sorts. It then expands to an application of the qualified predicate symbol to the fully-qualified expansions of the argument terms for those sorts.

```
DEFINEDNESS   ::=  definedness TERM
```

A definedness formula is well-sorted when the term is well-sorted for some sort. It then expands to a definedness assertion on the explicitly-sorted expansions of the term.

```
EQUATION      ::=  equation QUALITY TERM TERM
QUALITY       ::=  existential | strong
```

An equation is well-sorted if there is a sort such that both the terms are well-sorted for that sort. It then expands to the corresponding equation on the explicitly-sorted expansions of the terms for that sort.

```
TERM            ::=   VAR | APPLICATION | SORTED-TERM
```

A variable VAR is well-sorted for the sort specified in its declaration. It expands to the variable itself.

```
APPLICATION     ::=   application FUN-SYMB TERM*
FUN-SYMB        ::=   fun-symb FUN-NAME FUN-TYPE?
```

An application of a function symbol FUN-SYMB is well-sorted when there is a declaration of FUN-SYMB (with the specified FUN-TYPE, if any) such that all the argument terms are well-sorted for the declared argument sorts. It then expands to an application of the qualified function symbol to the fully-qualified expansions of the argument terms for those sorts.

```
SORTED-TERM     ::=   sorted-term TERM SORT
```

A sorted term is well-sorted if the component term is well-sorted for the specified sort. It then expands to those of the explicitly-sorted expansions of the component term that have the specified sort.

2.4 Sort Generation

```
SORT-GEN        ::=   sort-gen SORT+ FUN-SYMB+
```

A sort-generation constraint SORT-GEN is well-formed when all the sorts and function symbols specified in it are declared in the local environment (it does not itself declare any sorts and function symbols). It then determines the corresponding sentence. Here, a FUN-SYMB that is just a FUN-NAME (without an explicit FUN-TYPE) abbreviates the list of all those declared function symbols whose name has the same identifier as the specified FUN-NAME.

3 Subsorting Concepts

This section introduces the signatures, models, and sentences characterizing basic specifications with subsorts, extending Section 1. The notions of satisfaction and proof system for subsorted specifications are essentially as for many-sorted specifications.

The intuition behind the treatment of subsorts adopted here is to represent subsorting by embeddings (which are not required to be identities), commuting with overloaded function symbols, as usual in order-sorted approaches. However, in the language described in Section 4, instead of imposing conditions such as 'regularity' on signatures, terms and sentences that can be given different parses (up to the commutativity between embeddings and overloaded symbols) are simply rejected as ill-formed.

While *well-formedness* of terms of the language can be checked statically, the question whether the value of a well-formed term is actually defined or not may depend on the specified axioms (and it is not decidable in general).

3.1 Signatures

A *subsorted signature* $\Sigma = (S, TF, PF, P, \leq_S)$ consists of a many-sorted signature (S, TF, PF, P) together with a pre-order $\leq_S$ of *subsort embeddings* on the set S of sorts. $\leq_S$ is extended to sequences of sorts. We may drop the subscript S when obvious from the context.

For a subsorted signature, we can define *overloading relations* for function and predicate symbols. Let $f_1 : w_1 \to s_1 \leq_F f_2 : w_2 \to s_2$ hold if $w_1 \leq w_2$, s_1 and s_2 have a common supersort, and $f_1 = f_2$; similarly for predicate symbols, let $p_1 : w_1 \leq_P p_2 : w_2$ hold if $w_1 \leq w_2$ and $p_1 = p_2$.

A *subsorted signature morphism* $\sigma : \Sigma \to \Sigma'$ is a many-sorted signature morphism that preserves the pre-order $\leq_S$ and the overloading relations $\leq_F$ and $\leq_P$.

With each subsorted signature $\Sigma = (S, TF, PF, P, \leq_S)$ a many-sorted signature $\Sigma^{\#}$ is associated, extending (S, TF, PF, P) for each pair of sorts $s \leq_S s'$ by a total embedding function (from s into s'), a partial projection function (from s' onto s), and a membership predicate (testing whether values in s' are embeddings of values in s). We assume that the symbols used for embeddings, projections and membership are not used otherwise in Σ.

3.2 Models

For a subsorted signature Σ the *subsorted models* are ordinary many-sorted models for $\Sigma^{\#}$ that satisfy the following properties (which can be formalized as a set of conditional axioms):

- Embeddings are 1-1 functions.
- The embedding of a sort into itself is the identity function.
- All compositions of embeddings between the same two sorts are equal functions.
- Projections are the partial inverses of embeddings.
- Membership in a subsort holds just when the projection to the subsort is defined.

- Embeddings commute with those functions and predicates that are in the overloading relations.

3.3 Subsorted Sentences

For a subsorted signature Σ, the *subsorted sentences* are the ordinary many-sorted sentences (as defined in Section 1) for the associated many-sorted signature $\Sigma^{\#}$.

4 Subsorting Constructs

This section provides the abstract syntax and determines the intended interpretation of the constructs on *subsorted* basic specifications, extending what was provided for many-sorted specifications in Section 2.

A well-formed subsorted basic specification `BASIC-SPEC` of the CASL language determines a basic specification of the underlying subsorted institution, consisting of a subsorted signature and a set of sentences of the form described in Section 3. The models of this signature and set of sentences provide the semantics of the basic specification.

```
BASIC-ITEM      ::=   ... | SUBSORT-DECL | PRED-SORT-DEFN
```

A subsort declaration `SUBSORT-DECL` determines a contribution to the subsort embedding pre-order of a subsorted signature. A predicative subsort definition `PRED-SORT-DEFN` determines a subsort and its embedding in another sort, together with a sentence that characterizes the values of the subsort.

4.1 Subsort Declarations

```
SIG-DECL        ::=   ... | SUBSORT-DECL
SUBSORT-DECL    ::=   EMBEDDING-DECL | ISO-DECL
EMBEDDING-DECL  ::=   embedding-decl SORT-LAYER+
SORT-LAYER      ::=   sort-layer SORT+
ISO-DECL        ::=   SORT-LAYER
```

The sorts in a subsort declaration `SUBSORT-DECL` have to be already declared separately in the local environment, as such a construct merely determines a contribution to the subsort embedding pre-order of a subsorted signature. The same sort may not occur more than once in each `SUBSORT-DECL`.

By specifying an EMBEDDING-DECL, each sort in a SORT-LAYER is embedded in all the sorts in each SORT-LAYER on its right. Thus, an embedding declaration of the form:

embedding-decl... (sort-layer... s...)... (sort-layer... s'...)...

provides the embedding relation $s \leq s'$. An EMBEDDING-DECL is well-formed only when it has at least two SORT-LAYER components.

By specifying an ISO-DECL, each sort in its single SORT-LAYER is embedded in all the other sorts in that SORT-LAYER, thus allowing mutual and cyclic embeddings, which correspond to isomorphisms between carrier sets. An ISO-DECL is well-formed only when there are at least two SORT components in the SORT-LAYER.

Introducing an embedding relation between two sorts may cause function symbols to become related by the overloading relation, so that values of terms become equated when they are identical up to embeddings.

4.2 Predicative Subsort Definitions

```
BASIC-ITEM      ::=   ... | PRED-SORT-DEFN
PRED-SORT-DEFN  ::=   pred-sort-defn SORT VAR SORT FORMULA
```

A predicative sort definition PRED-SORT-DEFN determines the values of a new subsort of a previously-declared sort, by taking just those values for which a formula holds. It provides an explicit specification of the values of a subsort, in contrast to the implicit specification provided by using subsort declarations and overloaded function symbols.

The PRED-SORT-DEFN declares the first SORT; it declares the embedding of this as a subsort of the second SORT, which must already be declared in the local environment; and it asserts that the values of the subsort are precisely (the projections of) those values of the variable VAR from the supersort for which the FORMULA holds.

The variable VAR must occur in the FORMULA, and its scope is restricted to the FORMULA. Any other variables occurring in the FORMULA must be explicitly declared by enclosing quantifications.

Note that the only terms of the supersort that are also in the subsort are explicit projections, expressed using the CAST construct (described in the next section).

4.3 Axioms and Terms

As for many-sorted specifications, an atomic formula is well-formed (with respect to the current declarations) if it is well-sorted and expands to a unique atomic formula for constructing sentences of the underlying institution—but now for subsorted specifications, uniqueness is required only up to an *equivalence* on atomic formulae and terms. Two expansions t, t' of the same term are regarded as equivalent in the following cases:

- t' is the application of an embedding of t into a supersort of its sort;
- t and t' are the applications of different compositions of embeddings of the same term into the same sort;
- t' is the same as t up to profiles of function symbols in the overloading relation $\leq_F$ and embeddings.

Two expansions of the same atomic formula are regarded as equivalent in the following cases:

- the formulae are identical, up to replacement of equivalent terms; or
- the formulae are the same up to the profiles of predicate symbols in the overloading relation $\leq_P$ and embeddings.

The notions of when an atomic formula or term is *well-sorted* and of its *expansion* are indicated below for the various subsorting constructs. Due not only to overloading of predicate and/or function symbols, but also to implicit embeddings from subsorts into supersorts, a well-sorted atomic formula or term may have several expansions, preventing it from being well-formed. Qualifications on function and predicate symbols, or explicit sorts on terms, may be used to determine the intended expansion (up to the equivalence indicated above) and make it well-formed.

```
ATOM          ::=  ... | MEMBERSHIP
MEMBERSHIP    ::=  membership TERM SORT
```

Atomic formulae `ATOM` are extended with a construct for testing membership in subsorts. A membership formula is well-sorted if the specified `TERM` is well-sorted for a supersort s' of the specified `SORT` s. It expands to an application of the pre-declared predicate symbol for testing s' values for membership in the embedding of s.

```
TERM          ::=  ... | CAST
CAST          ::=  cast TERM SORT
```

Term formation is extended too, by providing a casting construct `CAST` for projections (there are no implicit casts). The cast of the specified `TERM` to a `SORT` s is well-sorted if the term is well-sorted for a supersort s' of s. It expands to an application of the pre-declared function symbol for projecting from s' to s.

Term formation is also extended by letting a term of a subsort s be regarded as a well-sorted term of a supersort s' as well, which provides implicit embeddings. It expands to the explicit application of the pre-declared function symbol for embedding s into s'. Also a `SORTED-TERM` expands to an explicit application of an embedding, provided that the apparent sort of the component `TERM` is a subsort of the specified `SORT`.

Part II

Structured Specifications

5 Structuring Concepts

A basic specification, as described in Part I, consists essentially of a signature Σ (declaring symbols) and a set of sentences (axioms or constraints) over Σ. The semantics of a well-formed basic specification is the specified signature signature Σ together with the class of all Σ-models that satisfy the specified sentences.

A *structured specification* is formed by combining specifications in various ways, starting from basic specifications. For instance, specifications may be *united*; a specification may be *extended* with further signature and/or sentences; parts of a signature may be *hidden*; the signature may be *translated* to use different symbols (with corresponding translation of the sentences) by a signature morphism; and models may be restricted to *initial* models. Such ways of combining specifications are independent of the details of the particular institution underlying basic specifications (so they can be applied to subsorted specifications just as well as to many-sorted specifications). The abstract syntax of constructs in the CASL language for presenting such structured specifications is described in Section 6.

The semantics of a well-formed structured specification is of the same form as that of a basic specification: a signature Σ together with a class of Σ-models. Thus the structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style. (Specification of the *architecture* of models in the CoFI framework is addressed in Part III.)

Within a structured specification, the *current signature* may vary more than in a basic specification. For instance, when two specifications are united, the signature valid in the one is generally different from that valid in the other. The association between symbols and their declarations as given by the valid signature is called the *local environment*. The *pre-declared symbols* (such as the constant predicates `true` and `false`) are always in the local environment, and may not be hidden or renamed.

The semantics of structured specifications involve signature morphisms and the corresponding reducts on models. For instance, hiding some symbols in a specification corresponds to a signature morphism that injects the non-hidden symbols into the original signature; the models, after hiding the symbols, are the reducts of the original models along this morphism.

Parts of structured specifications, in contrast to arbitrary parts of basic specifications, are potentially reusable—either verbatim, or with the adjustment of some *parameters*. Specifications may be *named*, so that the reuse of a specification may be replaced by a *reference* to it through its name. (Libraries of named specifications are explained in Part IV.) The current association between names and the specifications that they reference is called the *global environment*.

Named specifications that have some declared parameters are called *generic*. A reference to a generic specification should provide, for each parameter, an *argument specification* that *instantiates* it via a so-called *fitting morphism*. Under suitable restrictions, the union of the arguments, together with the translation of the generic specification by an expansion of the fitting morphism, corresponds to a so-called push-out construction.

6 Structuring Constructs

This section describes the abstract syntax and intended interpretation of the language constructs for structured specifications, extending that given for basic specifications in Part I.

The summary below indicates when structured specifications are well-formed, and how their signatures and classes of models are determined by those of their component specifications. Thus the interpretation is essentially based on models, rather than on a reduction to sets of sentences.

A structured specification can only be well-formed when all its component specifications are well-formed.

6.1 Structured Specifications

```
SPEC            ::=   BASIC-SPEC | TRANSLATION | REDUCTION
                |     UNION | EXTENSION | FREE-SPEC | TYPE-DEFN-GROUP
```

A structured specification `SPEC` may occur in a context where it is required to be *self-contained*, depending at most on the pre-declared symbols in the local environment (see the previous section for an explanation of the concept of environment used here). In that case, it determines a signature and a class of models straightforwardly. However, it may also occur in a context where it is supposed to *extend* other specifications, providing itself only part of a signature. Then it is interpreted as a function mapping signatures to the corresponding extended signatures, and from model classes over the former signatures to model classes over the extended signatures.

6.1.1 Translations and Reductions

```
TRANSLATION     ::=   translation SPEC SIG-MORPH
```

A translation is well-formed when the signature morphism construct SIG-MORPH expands to a signature morphism applicable to the signature of the component specification SPEC. The translation determines the signature obtained by applying the signature morphism, and the class of models consisting exactly of those models whose reducts along the morphism are among those of SPEC.

```
SIG-MORPH       ::=   sig-morph SYMB-MAP*
SYMB-MAP        ::=   SORT-MAP | FUN-SYMB-MAP | PRED-SYMB-MAP
SORT-MAP        ::=   sort-map SORT SORT
FUN-SYMB-MAP    ::=   fun-symb-map FUN-SYMB FUN-SYMB
PRED-SYMB-MAP   ::=   pred-symb-map PRED-SYMB PRED-SYMB
```

A signature morphism construct SIG-MORPH expands to a signature morphism applicable to a particular signature when each first component of a map in the list SYMB-MAP* is a symbol from that signature. Moreover, the union of the maps has to yield a function. Overloaded function symbols FUN-SYMB and predicate symbols PRED-SYMB may be disambiguated by explicit types; when no explicit type is given in a first component of a map SYMB-MAP, all function or predicate symbols with the same name are mapped uniformly.

```
REDUCTION       ::=   reduction RESTRICTION SPEC
RESTRICTION     ::=   restriction EXPOSURE SYMB+
EXPOSURE        ::=   hiding | revealing
SYMB            ::=   SORT | FUN-SYMB | PRED-SYMB
```

A REDUCTION is well-formed when the set of symbols determined by the RESTRICTION is a subset of the symbols declared in the signature of the component specification SPEC. In the case that the specified EXPOSURE is hiding, it determines the largest sub-signature of the signature of the component specification that does not contain these symbols (note that removing a sort entails removing all the functions and predicate symbols whose declared type involves that sort). In the case that the EXPOSURE is revealing, it determines the smallest sub-signature that does contain the indicated symbols (so revealing a function or predicate symbol entails revealing the sorts involved in its declared type). In both cases, the subsort embedding relation is inherited from that declared by SPEC, and the determined class of models is given by the reducts of the models of the component specification SPEC along the inclusion of the sub-signature determined by the RESTRICTION.

6.1.2 Unions and Extensions

```
UNION          ::=  union SPEC+
```

A UNION of several specifications abbreviates their joint (non-persistent) extension by an empty specification. The signature is obtained by the ordinary union of the signatures of each SPEC (not the disjoint union). The models are those whose reduct along each of the signature inclusion morphisms is a model of the respective SPEC.

```
EXTENSION       ::=  extension OF-SPEC* SPEC
OF-SPEC         ::=  PERSISTENT-SPEC | SPEC
PERSISTENT-SPEC ::=  persistent-spec SPEC
```

An EXTENSION allows the extension of the union of some specifications with further signature and sentences. For each PERSISTENT-SPEC used as an OF-SPEC, the reducts of the models of the EXTENSION along the corresponding signature inclusion must give the same class of models as that determined by the SPEC of the PERSISTENT-SPEC, otherwise the EXTENSION is deemed to be inconsistent.

All the OF-SPECs in an EXTENSION get the current local environment; the component SPEC gets moreover all the symbols declared by the OF-SPECs.

```
FREE-SPEC       ::=  free-spec SPEC
```

By using a FREE-SPEC as the component SPEC of an EXTENSION, the class of models is that given by the free extension. When a FREE-SPEC is used other than in an extension, it abbreviates the free extension of an empty OF-SPEC list, which restricts the models to the initial models of the component specification.

6.2 Type Definition Group

```
BASIC-ITEM     ::=  ... | TYPE-DEFN-GROUP
TYPE-DEFN-GROUP::=  type-defn-group TYPE-DEFN+ AXIOM*
```

A type definition group TYPE-DEFN-GROUP provides a concise way of specifying a collection of sorts equipped with function symbols called constructors and selectors. The intended interpretation is as a freely-generated extension: terms built with the constructors evaluate to distinct values, unless their equality is a consequence of the specified axioms.

The order of the type definitions TYPE-DEFN in a TYPE-DEFN-GROUP is insignificant.

```
TYPE-DEFN       ::=   type-defn SORT ALTERNATIVE+
ALTERNATIVE     ::=   CONSTRUCT | SORT
```

A type-definition `TYPE-DEFN` declares its component `SORT`. For each alternative `CONSTRUCT` it declares the required constructor and selector functions, and determines axioms asserting the expected relationship between any selectors and the constructors. It declares each `SORT` that occurs as an `ALTERNATIVE` to be a subsort of the `SORT` of the enclosing `TYPE-DEFN`. All sorts used in each `ALTERNATIVE` must be declared in the enclosing `TYPE-DEFN-GROUP` or in the local environment provided to it.

When each `ALTERNATIVE` in a `TYPE-DEFN` is a `SORT`, the defined type corresponds to a sort union (which need not be a disjoint union).

```
CONSTRUCT       ::=   construct FUN-NAME COMPONENTS*
```

A function `FUN-NAME` specified in a `CONSTRUCT` is declared as a `total` constructor. Each `COMPONENTS` specifies one or more argument sorts for the constructor; its result sort is the `SORT` of the enclosing `TYPE-DEFN`.

When all the `ALTERNATIVE` constructs are constants without components, the defined type corresponds to an (unordered) enumeration type.

```
COMPONENTS      ::=   components FUN-NAME* SORT
```

A `COMPONENTS` with an empty list `FUN-NAME*` of selector functions merely provides its `SORT` as an argument sort for the constructor for the enclosing `CONSTRUCT`.

A `COMPONENTS` with a list `FUN-NAME*` of length $n > 0$ provides n copies of its `SORT` as argument sorts for the constructor for the enclosing `CONSTRUCT`. Moreover, it declares each function `FUN-NAME` as a selector, with the argument sort given by the enclosing `TYPE-DEFN`, and with the indicated result sort. The function is declared as `total` if there is only one `CONSTRUCT` in the enclosing `TYPE-DEFN`, otherwise as `partial`.

A function symbol may be used more than once in a type definition group, giving rise to overloading.

```
BASIC-ITEM      ::=   ... | TYPE-DEFN-GROUP
```

When a `TYPE-DEFN-GROUP` occurs as a `BASIC-ITEM` in a `BASIC-SPEC`, it corresponds to an `EXTENSION` of the `TYPE-DEFN-GROUP` by the rest of the `BASIC-SPEC`. Regarding visibility of symbols, however, basic items that occur before the `TYPE-DEFN-GROUP` are not allowed to refer to the symbols declared by it.

6.3 Generic Specifications

```
SPEC-DEFN       ::=   spec-defn SPEC-NAME GEN-SPEC
SPEC-NAME       ::=   SIMPLE-ID
```

A specification definition SPEC-DEFN provides a name SPEC-NAME for a (possibly generic) specification GEN-SPEC.

```
GEN-SPEC        ::=   gen-spec OF-SPEC* SPEC
```

The abstract syntax of a generic specification GEN-SPEC is similar to those of an EXTENSION: each declared parameter is an OF-SPEC, so persistency may be required by use of a PERSISTENT-SPEC, and freeness of the body SPEC may be specified by use of a FREE-SPEC. The well-formedness and semantics of GEN-SPEC are essentially as for the corresponding EXTENSION, the only difference being that a fitting argument has to be supplied for each declared parameter when referencing a GEN-SPEC through its name, as explained below. A GEN-SPEC with an empty OF-SPEC* list may be regarded as an ordinary, non-generic specification.

N.B. When a declared parameter is merely a SPEC-NAME, it refers to an *existing* specification definition in the global environment—it is *not* a local name for an argument specification. A declared parameter, whatever its form, essentially just specifies the signature and sentences that the body of the generic specification may assume to be provided by the corresponding argument specification that is supplied when referencing the generic specification. Thus the parameter specification corresponds more to the type of the argument than to a formal parameter.

```
SPEC            ::=   ... | SPEC-INST
SPEC-INST       ::=   spec-inst SPEC-NAME FITTING-ARG* SIG-MORPH?
FITTING-ARG     ::=   fitting-arg SPEC SIG-MORPH?
```

A specification instantiation SPEC-INST refers to the possibly-generic specification named SPEC-NAME in the global environment, providing a fitting argument FITTING-ARG for each declared parameter (in the same order).

A fitting argument FITTING-ARG fits the argument specification SPEC to the corresponding parameter specification via a signature morphism SIG-MORPH. An omitted SIG-MORPH construct is taken to be the identity morphism.

When there is more than one parameter, the fitting argument morphisms have to be *compatible*, with all of them extending to a single morphism, including moreover any fitting morphism SIG-MORPH that is specified as a component of the enclosing SPEC-INST. Thus any common parts of declared parameters have to be instantiated in the same way. It is pointless to declare the same parameter twice in a generic specification. (Generic specifications that require two similar but independent parameters can be expressed by using a translation to distinguish between the symbols in the signatures of the two parameters.)

The semantics of a well-formed instantiation `SPEC-INST` is the same as that of the specification determined as follows: the fitting morphisms are extended to a single morphism applicable to the signature of the entire generic specification `GEN-SPEC` named by the `SPEC-NAME`, any persistency and freeness requirements are checked, then the translation of it by this morphism is united with all the argument specifications.

```
ID              ::=   ... | COMPOUND-ID
COMPOUND-ID     ::=   compound-id SIMPLE-ID ID+
```

This extension of the syntax of identifiers for sorts, functions, and predicates is of relevance to generic specifications. In a compound identifier `COMPOUND-ID`, the components may (but need not) themselves identify sorts, functions, or predicates that are specified in the declared parameters of a generic specification.

When such a compound identifier is used to name, e.g., a sort in the body of a generic specification, the translation determined by fitting arguments to parameters applies to the components `ID+` of the compound identifier as well. Thus instantiations with different arguments generally give rise to different compound identifiers for what would otherwise be the same sort, which avoids unintended identifications when the instantiations are united.

E.g., a generic specification of sequences of arbitrary elements might use the simple identifier `Elem` for a sort in the parameter, and a compound identifier `compound-id Seq Elem` for the sort of sequences in the body. Fitting various argument sorts to `Elem` in different instantiations then results in distinct sorts of sequences.

Subsort embeddings between component sorts induce subsort embeddings between the compound sorts. For example, when `Nat` is declared as a subsort of `Int`, we get automatically `(compound-id Seq Nat)` embedded as a subsort of `(compound-id Seq Int)` in signatures containing all these sorts.

The details of the intended relationship between compound identifiers and other constructs (such as translations and reductions) have still to be decided. Higher-order extensions of CASL are expected to provide a more semantic treatment of parametrized sorts, etc.

Part III

Architectural Specifications

7 Architectural Concepts

The intention with architectural specifications is primarily to impose structure on models, expressing their *composition* from component units—and thereby also a *decomposition* of the task of developing such models from requirements specifications. This is in contrast to the structured specifications summarized in Part II, where the specified models have no more structure than do those of the basic specifications summarized in Part I.

The component units may all be regarded as *unit constructor functions*: functions without arguments give self-contained units; functions with arguments use such units in constructing further units. Note that a resulting unit may be needed for use as an argument in more than one application.

The specification of a unit constructor function indicates the properties to be assumed of the arguments, and the properties to be guaranteed of the result. Such a specification provides the appropriate interfaces for the development of the function. In **CASL**, self-contained units are simply algebras, and their properties are expressed by ordinary (perhaps structured) specifications.

Thus a unit constructor function maps models of argument specifications to models of a result specification. A specification of such functions can be simply a list of the argument specifications together with the result specification. Thinking of argument and result specifications as *types* of algebras, a specification of a unit constructor function may be regarded as a *function type*.

An entire *architectural specification* is a collection of unit function specifications, together with a description of how the functions are to be composed to give a resulting unit. A model of an architectural specification is a collection of unit functions, together with a particular way of composing them.

The intention is that a unit function should actually make use of its arguments. In particular, it should not re-implement the argument specifications. This is ensured by requiring the unit function to be *persistent*: the reduct of the result to each argument signature yields exactly the given arguments.

As a consequence, the result *signature* has to include each argument *signature*—any desired hiding has to be left to when functions are composed. Moreover, since each *symbol* in the union of the argument signatures has to be implemented the same way in the result as in each argument where it occurs, the arguments must already have the same implementation of all common symbols. Let us call such arguments *compatible*.

Hence the interpretation of the specification of a unit function is as all *persistent* functions from *compatible* tuples of models of the argument specifications to models of the result specification. When composing such functions, care must be taken to ensure that arguments are indeed compatible. Notice that if two arguments have the same specification, the corresponding arguments must be identical. It is not possible to specify a function that should take two arguments that implement the same signature independently—although one can get the same effect, by renaming one or both of the argument signatures.

8 Architectural Constructs

This section describes the abstract syntax and determines the intended interpretation of the language constructs for architectural specifications, extending that given for basic and structured specifications in Parts I and II.

```
ARCH-SPEC-DEFN ::=  arch-spec-defn SPEC-NAME ARCH-SPEC
ARCH-SPEC      ::=  arch-spec UNIT-DECL+ RESULT-UNIT
```

An architectural specification definition `ARCH-SPEC-DEFN` provides a name `SPEC-NAME` for an architectural specification `ARCH-SPEC`, which consists of a list of unit declarations `UNIT-DECL+`, together with a description of how such units are to be composed in a `RESULT-UNIT`. A model of an architectural specification consists of a unit for each `UNIT-DECL`, and the described composition of these units.

8.1 Unit Declarations

```
UNIT-DECL      ::=  unit-decl UNIT-NAME UNIT-SPEC
UNIT-NAME      ::=  SIMPLE-ID
```

Each unit declaration `UNIT-DECL` provides not only a unit specification `UNIT-SPEC` but also a `UNIT-NAME`, which is used for referring to the unit in the specification of the result unit. The same unit may be used more than once in the composition. No `UNIT-NAME` may be declared more than once in a particular architectural specification.

```
UNIT-SPEC-DEFN ::=  unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC      ::=  UNIT-TYPE | SPEC-NAME
UNIT-TYPE      ::=  unit-type SPEC* SPEC
```

A unit specification definition UNIT-SPEC-DEFN provides a name SPEC-NAME for a unit specification UNIT-SPEC, which is either a UNIT-TYPE or the name SPEC-NAME of another unit specification. A unit satisfies a UNIT-TYPE when it is a persistent function that maps compatible tuples of models of the argument specifications SPEC* to models of their extension by the result specification SPEC. A UNIT-TYPE is well-formed only when the signature of the union of the argument specifications is included in the signature of the result.

8.2 Unit Compositions

```
RESULT-UNIT     ::=  result-unit UNIT-DECL* UNIT-TERM
UNIT-TERM       ::=  UNIT-APPL | UNIT-REDUCT
UNIT-APPL       ::=  unit-appl UNIT-NAME UNIT-TERM*
UNIT-REDUCT     ::=  unit-reduct SIG-MORPH UNIT-TERM
```

A RESULT-UNIT determines how the units declared in the enclosing architectural specification are to be composed, using a UNIT-TERM built from applications UNIT-APPL. A unit reduct UNIT-REDUCT allows parts of the resulting unit to be hidden or renamed. Taking the UNIT-TYPE of each UNIT-NAME from its declaration, and taking any reducts into account, the UNIT-TERM must be well-typed: the signature of each argument UNIT-TERM must match that of the corresponding argument specification, and the compatibility of the arguments must be ensured. The auxiliary unit declarations UNIT-DECL* in a RESULT-UNIT are for (non-parametrized) units that are required, but not specified, by the enclosing architectural specification; this allows compositions which express partial architectural specifications that depend on external units as specified in the declarations.

Part IV

Specification Libraries

9 Library Concepts

Specifications may be *named* by *definitions* and collected in *libraries*. In the context of a library, the (re)use of a specification may be replaced by a *reference* to it through its name. The current association between names and the specifications that they reference is called the *global environment*; it may vary throughout a library, e.g., with *linear visibility* the global environment for a named specification is determined exclusively by the definitions that precede it.

With *referential transparency*, a reference to a named specification may always be replaced by a copy of the specification itself. Thus in principle, all references to names could be eliminated from the specifications in a library—provided that there are no mutual or cyclic chains of references, i.e., the library can be linearized.

The local environment given to each named specification in a library should be independent of the other specifications in the library. This allows the well-formedness of each named specification to be independent of the references made to it.

A library may be located at a particular *site* on the Internet, and addressed by a *URL* (uniform resource locator). Each library may incorporate the *downloading* of named specifications from other libraries, which takes place whenever the library is used.

The semantics of a specification library is a map taking each specification name defined in it to the semantics of that specification. The initial global environment for the library is determined by the semantics of any downloaded specifications.

10 Library Constructs

This section describes the abstract syntax and determines the intended interpretation of the language constructs for specification libraries, extending that given for basic, structured, and architectural specifications in Parts I–III.

10.1 Libraries

```
LIBRARY         ::=  library LIBRARY-ITEM*
LIBRARY-ITEM    ::=  SPEC-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
```

A LIBRARY provides a collection of specification definitions, possibly of various kinds. It is well-formed only when each SPEC-NAME that occurs in the LIBRARY is uniquely defined, and not referenced until (strictly) after its definition. The global environment for each named specification is that determined by the preceding definitions. Thus a LIBRARY here provides linear visibility, and mutual or cyclic chains of references are not allowed.

The local environment for each named specification is fixed, consisting only of the pre-declared signature (the constant predicates true and false). Each SPEC defined by a library must be self-contained, determining a complete signature after resolving references to names defined in the current global environment—fragments of specifications cannot be named.

A LIBRARY determines a map from names to the semantics of the named specifications.

10.2 Downloading

```
LIBRARY         ::=  ... | library URL? LIBRARY-ITEM*
LIBRARY-ITEM    ::=  ... | DOWNLOAD
DOWNLOAD        ::=  download URL SPEC-NAME-MAP+
SPEC-NAME-MAP   ::=  spec-name-map SPEC-NAME? SPEC-NAME
URL             ::=  url SITE? DIRECTORY
SITE            -- structure insignificant for abstract syntax
```

A DOWNLOAD specifies which definitions to copy from a remote library, with possible renaming as indicated by the optional SPEC-NAME? in a SPEC-NAME-MAP. Any unmentioned remote definitions to which the downloaded definitions refer are copied too, but their names are not added to the global environment. Thus a DOWNLOAD shows exactly which specification names are added to the initial global environment of the LIBRARY in which it occurs, allowing references to named specifications in the LIBRARY to be checked before copying the definitions from the remote library.

Cyclic or self-referential downloads are not allowed.

```
URL             ::=  url SITE? DIRECTORY
SITE            -- structure insignificant for abstract syntax
```

The DIRECTORY of a URL identifies the location of the library definitions at the specified SITE. When the SITE is omitted, the DIRECTORY is interpreted relative to the URL specified in the enclosing LIBRARY.

Appendix A: Abstract Syntax

The *abstract syntax* is central to the definition of a formal language. It stands between the concrete representations of documents, such as marks on paper or images on screens, and the abstract entities, semantic relations, and semantic functions used for defining their meaning.

The abstract syntax has the following objectives:

- to identify and separately name the abstract syntactic entities;
- to simplify and unify underlying concepts, putting like things with like, and reducing unnecessary duplication.

There are many possible ways of constructing an abstract syntax, and the choice of form is a matter of judgement, taking into account the somewhat conflicting aims of simplicity and economy of semantic definition.

The abstract syntax is presented as a set of production rules in which each entity is defined in terms of its constituent parts. The productions form a context-free grammar. The notation `X*`, `X+`, `X?` indicates the repetition of `X` any number of times, at least once, and at most once, respectively. The context conditions for well-formedness of specifications are not determined by the grammar (these are to be defined separately).

For each of the entities defined in the abstract syntax, there will eventually be a subsection in the reference manual describing its representation and meaning. To obtain a clear relationship with the concrete representation, the constructors used to distinguish the alternatives may later be replaced by more suggestive symbols. The order in which components of constructs are currently listed does not necessarily correspond to that to be used in the concrete representation.

The current emphasis of the abstract syntax is on documenting the individual constructs, and on fixing terminology. The internal organization of the abstract syntax will almost certainly need adjusting before it is optimal for use in connection with semantics or tools. Note that an abstract syntax is provided for the entire language, not just for a semantic kernel (but the semantics of constructs that are essentially just abbreviations or 'syntactic sugar' may still be defined by expansion).

Identifiers

```
ID            ::=   SIMPLE-ID
SIMPLE-ID     -- structure insignificant for abstract syntax
```

Basic Specifications

```
BASIC-SPEC      ::=  basic-spec BASIC-ITEM*
BASIC-ITEM      ::=  SIG-DECL | VAR-DECL | AXIOM | SORT-GEN

SIG-DECL        ::=  SORT-DECL | FUN-DECL | PRED-DECL
SORT-DECL       ::=  sort-decl SORT+
FUN-DECL        ::=  fun-decl  FUN-NAME+ FUN-TYPE
PRED-DECL       ::=  pred-decl PRED-NAME+ PRED-TYPE
FUN-TYPE        ::=  fun-type  TOTALITY SORT* SORT
TOTALITY        ::=  total | partial
PRED-TYPE       ::=  pred-type SORT*

VAR-DECL        ::=  var-decl VAR+ SORT

AXIOM           ::=  FORMULA
FORMULA         ::=  QUANTIFICATION | CONJUNCTION | DISJUNCTION
                 |   IMPLICATION | EQUIVALENCE | NEGATION | ATOM
QUANTIFICATION  ::=  quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER      ::=  forall | exists | exists-uniquely
CONJUNCTION     ::=  conjunction FORMULA+
DISJUNCTION     ::=  disjunction FORMULA+
IMPLICATION     ::=  implication FORMULA FORMULA
EQUIVALENCE     ::=  equivalence FORMULA FORMULA
NEGATION        ::=  negation FORMULA

ATOM            ::=  TRUTH | PREDICATION | DEFINEDNESS | EQUATION
TRUTH           ::=  true | false
PREDICATION     ::=  predication PRED-SYMB TERM*
DEFINEDNESS     ::=  definedness TERM
EQUATION        ::=  equation QUALITY TERM TERM
QUALITY         ::=  existential | strong

TERM            ::=  VAR | APPLICATION | SORTED-TERM
APPLICATION     ::=  application FUN-SYMB TERM*
SORTED-TERM     ::=  sorted-term TERM SORT

SORT-GEN        ::=  sort-gen SORT+ FUN-SYMB+

FUN-SYMB        ::=  fun-symb FUN-NAME FUN-TYPE?
PRED-SYMB       ::=  pred-symb PRED-NAME PRED-TYPE?

SORT            ::=  ID
FUN-NAME        ::=  ID
PRED-NAME       ::=  ID
VAR             ::=  SIMPLE-ID
```

Basic Specifications with Subsorts

```
SIG-DECL        ::=  ... | SUBSORT-DECL
SUBSORT-DECL    ::=  EMBEDDING-DECL | ISO-DECL
EMBEDDING-DECL  ::=  embedding-decl SORT-LAYER+
SORT-LAYER      ::=  sort-layer SORT+
```

```
ISO-DECL        ::=   SORT-LAYER

BASIC-ITEM      ::=   ... | PRED-SORT-DEFN
PRED-SORT-DEFN  ::=   pred-sort-defn SORT VAR SORT FORMULA

ATOM            ::=   ... | MEMBERSHIP
MEMBERSHIP      ::=   membership TERM SORT
TERM            ::=   ... | CAST
CAST            ::=   cast TERM SORT
```

Structured Specifications

```
SPEC            ::=   BASIC-SPEC | TRANSLATION | REDUCTION
                |     UNION | EXTENSION | FREE-SPEC | TYPE-DEFN-GROUP
TRANSLATION     ::=   translation SPEC SIG-MORPH
REDUCTION       ::=   reduction RESTRICTION SPEC
RESTRICTION     ::=   restriction EXPOSURE SYMB+
EXPOSURE        ::=   hiding | revealing
SYMB            ::=   SORT | FUN-SYMB | PRED-SYMB
UNION           ::=   union SPEC+
EXTENSION       ::=   extension OF-SPEC* SPEC
OF-SPEC         ::=   PERSISTENT-SPEC | SPEC
PERSISTENT-SPEC ::=   persistent-spec SPEC
FREE-SPEC       ::=   free-spec SPEC

SIG-MORPH       ::=   sig-morph SYMB-MAP*
SYMB-MAP        ::=   SORT-MAP | FUN-SYMB-MAP | PRED-SYMB-MAP
SORT-MAP        ::=   sort-map SORT SORT
FUN-SYMB-MAP    ::=   fun-symb-map FUN-SYMB FUN-SYMB
PRED-SYMB-MAP   ::=   pred-symb-map PRED-SYMB PRED-SYMB

BASIC-ITEM      ::=   ... | TYPE-DEFN-GROUP
TYPE-DEFN-GROUP ::=   type-defn-group TYPE-DEFN+ AXIOM*
TYPE-DEFN       ::=   type-defn SORT ALTERNATIVE+
ALTERNATIVE     ::=   CONSTRUCT | SORT
CONSTRUCT       ::=   construct FUN-NAME COMPONENTS*
COMPONENTS      ::=   components FUN-NAME* SORT
```

Generic Specifications

```
SPEC-DEFN       ::=   spec-defn SPEC-NAME GEN-SPEC
SPEC-NAME       ::=   SIMPLE-ID
GEN-SPEC        ::=   gen-spec OF-SPEC* SPEC

SPEC            ::=   ... | SPEC-INST
SPEC-INST       ::=   spec-inst SPEC-NAME FITTING-ARG* SIG-MORPH?
FITTING-ARG     ::=   fitting-arg SPEC SIG-MORPH?

ID              ::=   ... | COMPOUND-ID
COMPOUND-ID     ::=   compound-id SIMPLE-ID ID+
```

Architectural Specifications

```
ARCH-SPEC-DEFN  ::=   arch-spec-defn SPEC-NAME ARCH-SPEC
ARCH-SPEC       ::=   arch-spec UNIT-DECL+ RESULT-UNIT

UNIT-DECL       ::=   unit-decl UNIT-NAME UNIT-SPEC
UNIT-NAME       ::=   SIMPLE-ID

UNIT-SPEC-DEFN  ::=   unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC       ::=   SPEC-NAME | UNIT-TYPE
UNIT-TYPE       ::=   unit-type SPEC* SPEC

RESULT-UNIT     ::=   result-unit UNIT-DECL* UNIT-TERM
UNIT-TERM       ::=   UNIT-APPL | UNIT-REDUCT
UNIT-APPL       ::=   unit-appl UNIT-NAME UNIT-TERM*
UNIT-REDUCT     ::=   unit-reduct SIG-MORPH UNIT-TERM
```

Specification Libraries

```
LIBRARY         ::=   library URL? LIBRARY-ITEM*
LIBRARY-ITEM    ::=   SPEC-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
                |     DOWNLOAD
DOWNLOAD        ::=   download URL SPEC-NAME-MAP+
SPEC-NAME-MAP   ::=   spec-name-map SPEC-NAME? SPEC-NAME
URL             ::=   url SITE? DIRECTORY
SITE            -- structure insignificant for abstract syntax
```

Appendix B: Bibliography

Language Design Study Notes

The CoFI Language Design Study Notes[9] provide much background material for the tentative design decisions that have been taken.

For a more detailed motivation of the treatment of *subsorts* and a comparison with alternative treatments, see the study note [MC++-1][10], to be available shortly.

Recent notes on *generic specifications* [MB-2][11] and on *architectural specifications* [PDM-2][12] provided the basis for the tentative design of these constructs.

Catalogue of Specification Languages

A conscious attempt has been made to base the tentative design on constructs known from previous algebraic specification languages. A good number of these languages are summarized in the CoFI Catalogue[13].

Correlation between CASL and existing languages

In preparation.

Standard Examples of Specifications in CASL

To be written.

Rationale for Tentative Design

To be written, based on the existing Language Design Study Notes.

[9] http://www.brics.dk/Projects/CoFI/StudyNotes.html
[10] http://www.brics.dk/Projects/CoFI/StudyNotes/Lang/MC++-1.html
[11] http://www.brics.dk/Projects/CoFI/StudyNotes/Lang/MB-2.html
[12] http://www.brics.dk/Projects/CoFI/StudyNotes/Lang/PDM-2.html
[13] http://www.brics.dk/Projects/CoFI/Catalogue/4/index.html

Appendix C: Proposed Changes

This appendix lists some proposed changes (additions) to the tentative design. These were discussed at the Edinburgh meeting and proposed in the draft of the present document, but it was felt that some further investigation was needed before incorporating them. (This investigation had not been completed by the time this document was released.) These proposed additions are not expected to have any impact on other language constructs.

Basic Constructs: Attribution

```
BASIC-ITEM      ::=   ... | ATTRIBUTION
ATTRIBUTION     ::=   attribution PROPERTY+ FUN-SYMB+
PROPERTY        ::=   SIMPLE-PROPERTY | UNIT-PROPERTY
SIMPLE-PROPERTY ::=   associative | commutative | idempotent
UNIT-PROPERTY   ::=   unit-property FUN-SYMB
```

An ATTRIBUTION of some properties to some function symbols FUN-SYMB+ abbreviates evident axioms asserting that each of the functions is to have each of the specified properties. Here, a FUN-SYMB that is just a FUN-NAME (without an explicit FUN-TYPE) abbreviates the list of all those function symbols declared in the local environment whose name has the same identifier as the specified FUN-NAME. The ATTRIBUTION is well-formed just when the axioms that it abbreviates are well-formed—in particular, this generally requires that each specified function symbol takes two arguments, both of the same sort.

Basic Constructs: Local Declarations and Definitions

Local declarations are proposed in basic specifications; local definitions are proposed in formulae. In each case, the scope of the locally declared/defined items is the remainder of the phrase only.

```
BASIC-ITEM        ::=   ... | LOCAL-BASIC-ITEMS
LOCAL-BASIC-ITEMS ::=   local-basic-items BASIC-ITEM+ BASIC-ITEM*
```

The extension introduced by a series of local basic items is required to be persistent (otherwise the specification is deemed inconsistent). The scope of the basic items declared in the first list BASIC-ITEM+ is restricted to the second list of basic items, whereas the scope of those declared in the second list BASIC-ITEM* includes the following basic items of the enclosing BASIC-SPEC.

```
FORMULA       ::=   ... | LOCAL-VARS
LOCAL-VARS    ::=   local-vars VAR-DEFN+ FORMULA
VAR-DEFN      ::=   var-defn VAR SORT TERM
```

A variable definition VAR-DEFN has the same effect as declaring the variable VAR to have the specified SORT, and asserting that its value is equal to that of the TERM. The scope of the variables defined in VAR-DEFN+ is the following FORMULA only.

Architectural Constructs: Local Definitions

```
UNIT-TERM      ::=   ... | LOCAL-UNITS
LOCAL-UNITS    ::=   local-units UNIT-DEFN+ UNIT-TERM
UNIT-DEFN      ::=   unit-defn UNIT-NAME UNIT-TERM
```

(Alternatively:

```
UNIT-DEFN ::= unit-defn UNIT-NAME RESULT-UNIT
```

if bound units are allowed to depend on additional specified external units.)

Local definitions LOCAL-UNITS are proposed in unit terms. The scope of the names defined in each unit definition in UNIT-DEFN+ is the following unit definitions and the UNIT-TERM only. A UNIT-DEFN defines the UNIT-NAME as an abbreviation for the UNIT-TERM.

Library Constructs: Local Definitions

```
LIBRARY-ITEM        ::= ... | LOCAL-LIBRARY-ITEMS
LOCAL-LIBRARY-ITEMS ::= local-library-items LIBRARY-ITEM+ LIBRARY-ITEM*
```

Local definitions LOCAL-LIBRARY-ITEMS are proposed in libraries. The scope of each library item defined in the first list LIBRARY-ITEM+ is the following items of that list and the second list LIBRARY-ITEM* only. The scope of each library item defined in the second list is the following items of that list and the following library items of the enclosing LIBRARY.

Index

Recent Publications in the BRICS Notes Series

NS-96-15 **CoFI.** *CASL – The CoFI Algebraic Specification Language; Tentative Design: Language Summary*. **December 1996. viii+34 pp.**

NS-96-14 **Peter D. Mosses.** *A Tutorial on Action Semantics*. **December 1996. 46 pp. Tutorial notes for FME '94 (Formal Methods Europe, Barcelona, 1994) and FME '96 (Formal Methods Europe, Oxford, 1996).**

NS-96-13 **Olivier Danvy, editor.** *Proceedings of the Second ACM SIGPLAN Workshop on Continuations, CW '97* **(ENS, Paris, France, 14 January, 1997), December 1996. 166 pp.**

NS-96-12 **Mandayam K. Srivas.** *A Combined Approach to Hardware Verification: Proof-Checking, Rewriting with Decision Procedures and Model-Checking; Part II: Articles. BRICS Autumn School on Verification*. **October 1996. 56 pp.**

NS-96-11 **Mandayam K. Srivas.** *A Combined Approach to Hardware Verification: Proof-Checking, Rewriting with Decision Procedures and Model-Checking; Part I: Slides. BRICS Autumn School on Verification*. **October 1996. 29 pp.**

NS-96-10 **Robert Pollack.** *What we Learn from Formal Checking; Part III: Formalization is Not Just Filling In Details. BRICS Autumn School on Verification*. **October 1996. iv+42 pp.**

NS-96-9 **Robert Pollack.** *What we Learn from Formal Checking; Part II: Using Type Theory: An Introduction. BRICS Autumn School on Verification*. **October 1996. iv+71 pp.**

NS-96-8 **Robert Pollack.** *What we Learn from Formal Checking; Part I: How to Believe a Machine-Checked Proof. BRICS Autumn School on Verification*. **October 1996. iv+19 pp.**

NS-96-7 **Tom F. Melham.** *Some Research Issues in Higher Order Logic Theorem Proving. BRICS Autumn School on Verification*. **October 1996. 15 pp.**