**Basic Research in Computer Science**



# CASL: A Guided Tour of its Design

**Peter D. Mosses**

**BRICS Report Series** **RS-98-43**

**ISSN 0909-0878** **December 1998**

**See back inner page for a list of recent BRICS Report Series publications. 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 the World Wide Web and anonymous FTP through these URLs:**

`http://www.brics.dk`
`ftp://ftp.brics.dk`
**This document in subdirectory** `RS/98/43/`

# CASL: A Guided Tour of its Design

Peter D. Mosses[1,2]

[1] BRICS and Department of Computer Science,
University of Aarhus, Denmark
[2] CoFI: The Common Framework Initiative
for algebraic specification and development

**Abstract.** CASL is an expressive language for the specification of functional requirements and modular design of software. It has been designed by CoFI, the international Common Framework Initiative for algebraic specification and development. It is based on a critical selection of features that have already been explored in various contexts, including subsorts, partial functions, first-order logic, and structured and architectural specifications. CASL should facilitate interoperability of many existing algebraic prototyping and verification tools.
This guided tour of the CASL design is based closely on a 1/2-day tutorial held at ETAPS'98 (corresponding slides are available from the CoFI archives). The major issues that had to be resolved in the design process are indicated, and all the main concepts and constructs of CASL are briefly explained and illustrated—the reader is referred to the CASL Language Summary for further details. Some familiarity with the fundamental concepts of algebraic specification would be advantageous.

## 1 Background

The algebraic approach to software specification originated in the early 1970's. Since then, dozens of algebraic specification languages have been developed—all of them supporting the basic idea of using axioms to specify algebras, but differing in design choices concerning syntax (concrete and abstract) and semantics. The lack of a *common* framework for algebraic specification and development has discouraged industrial acceptance of the algebraic method, hindered its dissemination, and limited tool applicability.

Why not agree on a common framework? This was the provocative question asked at a WADT meeting in Santa Margherita, 1994. At least the main concepts to be incorporated were thought to be clear—although it was realized that it might not be so easy to agree on a common language to express these concepts.

The following aims and scope were formulated at the start of the Common Framework Initiative, CoFI, in September 1995 [13]:

The aims of CoFI are to provide a common framework:

- by a collaborative effort
- for algebraic specification and development
- attractive to researchers as well as for use in industry
- providing a common specification language with uniform, user-friendly syntax and straightforward semantics
- able to subsume many previous frameworks
- with good documentation and tool support
- free—but protected (cf. GNU)

The scope of CoFI is:

- specification of functional requirements
- formal development and verification of software
- relation of specifications to informal requirements and implemented code
- prototyping, theorem-proving
- libraries, reuse, evolution
- tool interoperability

The specification language developed by CoFI is called Casl: the Common Algebraic Specification Language. Its main features are:

- a critical selection of known constructs
- expressive, simple, pragmatic
- for specifying requirements and design for conventional software packages
- restrictions to sublanguages
- extensions to higher-order, state-based, concurrent, ...

The Casl design effort started in September 1995. An initial design was proposed in May 1997 (with a language summary, abstract syntax, formal semantics, but no agreed concrete syntax) and tentatively approved by IFIP WG1.3. Apart from a few details, the design was finalized in April 1998, with a complete draft language summary available, including concrete syntax. Casl version 1.0 was released in October 1998; the formal semantics given for the proposed design has now been updated to reflect the changes.

## 2 Guide

CASL consists of several major *parts*, which are quite independent and may be understood (and used) separately:

- basic specifications: declarations, definitions, axioms
- structured specifications: translations, reductions, unions, extensions, freeness, named specifications, generic specifications, views
- architectural specifications: implementation units, composition
- specification libraries: local, distributed

The above division of CASL into parts is orthogonal to taking sublanguages of CASL.

The CASL language design integrates several different *aspects*, which are here explained separately:

- pragmatic issues: methodology, tools, aesthetics
- semantic concepts: formal (institutions, environments), informal (expansions, scopes)
- language constructs: abstract syntax (structure, annotations); concrete syntax (input format, display format)

In this guided tour, each part of CASL is presented in turn, considering all the aspects listed above before proceeding to the next part. Of course the reader is free to ignore the guide, and wander through the various sections in a different order (but then the author cannot accept responsibility for any resulting disorientation...).

## 3 Basic Specifications

Basic specifications consist of declarations, definitions, and axioms. Section 3.1 considers various pragmatic issues affecting the CASL design. Section 3.2 presents the main concepts that underly the semantics of basic specifications. Finally, Section 3.3 provides examples that illustrate the CASL language constructs for use in basic specifications.

### 3.1 Pragmatic Issues

*Partial and Total Functions:* CASL supports both partial and total algebraic specification. Partiality is a particularly natural and simple

way of treating errors such as division by zero, and error propagation is implicit, so that whenever any argument of an operation is undefined, the result is undefined too. CASL also includes subsorts and error supersorts, to allow specification of exception handling when this is relevant. Totality is of course an important property, and CASL allows it to be declared along with the types of functions, rather than relegating it to the axioms. The domain of definition of a partial function may be made explicit by introducing it as a subsort of the argument sort and declaring the function to be total on it.

For instance, consider the familiar operations on (possibly-empty) lists: the list constructor *cons* would be declared as total, whereas the list head and tail selectors would be partial, being undefined on the empty list. The domain of definition of the selectors may be made explicit by introducing the subsort of non-empty lists, and declaring them to be total functions on that subsort. (This and further examples are specified in CASL in Section 3.3.)

In the presence of partiality, equations may require definedness: so-called 'existential' equations require it, 'strong' equations do not. In general, it is appropriate to use existential equations in conditions (since properties do not usually follow from undefinedness) but strong equations when defining partial functions inductively. So CASL allows both kinds of equations.

Definedness assertions can also be expressed directly. In fact definedness of a term is equivalent to its existential equality to itself—it could also be regarded as a unary predicate. Existential equality is equivalent to the conjunction of a strong equality and two definedness assertions; strong equality is equivalent to the conjunction of two conditionals involving only existential equality.

*Logic and Predicates:* CASL is based on classical 2-valued first-order logic, with the standard interpretation of connectives. It supports user-declared predicates, which have some advantages over the (total) Boolean functions that were used instead of predicates in most previous languages.[1] When an argument of a predicate is undefined, the predicate application cannot hold.

CASL provides the standard universal and existential quantification and logical connectives, i.e., ordinary first-order predicate logic.

[1] For example, predicates hold minimally in initial models.

The motivation for this is expressiveness: restricting to conditional equations sometimes requires quite contrived specifications. For instance, it is a straightforward exercise to specify when a string is a permutation of another using quantifiers, and negation provides the complementary property; but the latter is quite awkward to specify using (positive) conditional equations.

The usual equational and conditional specification frameworks are provided as sublanguages of CASL, simply by restricting the use of quantifiers and logical connectives.

*Classes of Models:* Most previous frameworks allow only one kind of model class to be specified. The default in CASL is to take all models of a specification, i.e., so-called loose semantics; but it is also possible to specify the restriction of models to the class of generated models (only expressible values are included, and properties may be proved by induction) or to the class of initial (and freely-generated) models (providing minimal satisfaction of atomic formulae). Of course initial models may not exist when disjunction, negation, etc., are used.

*Overloading:* In a CASL specification the same symbol may be declared with various profiles (i.e., list of argument and result sorts), e.g. '+' may be declared as an operation on integers, reals, and strings. When such an overloaded symbol is used, the intended profile is to be determined by the context. Explicit disambiguation can be used when needed, by specifying the profile (or result sort) in an application.

*Subsorts:* It is appropriate to declare a sort as a subsort of another when the values of the subsort are regarded a special case of those in the other sort. For instance, the positive integers and the positive odd integers are best regarded as subsorts of the sort of natural numbers, which is itself a subsort of the integers. In contrast to most previous frameworks, CASL interprets subsorts as *embeddings* between carriers—not necessarily inclusions. This allows, e.g., models where values of sort integer are represented differently from values of sort real (as in most computers). CASL still allows the models where the subsort happens to be a subset. The extra generality of embeddings seems to be useful, and does not complicate the foundations at all.

Subsort embeddings commute with overloaded functions, so the values are independent of which profiles are used: 2+2=4, regardless of whether the '+' is that declared for natural numbers or integers.

CASL does not impose any conditions of 'regularity', 'coherence', or 'sensibleness' on the relationship between overloading and subsorts. This is partly for simplicity (no such conditions are required for the semantics of CASL), partly because most such conditions lack modularity (which is a disadvantage in connection with structured specifications). Note that overloaded constants are allowed in CASL.

*Datatype Constructors/Selectors:* Specifications of 'datatypes' with constructor and (possibly also) selector operations are frequently needed: they correspond to (unions of) record types and enumeration types in programming languages. CASL provides special constructs for datatype declarations to abbreviate the rather tedious declarations and axioms for constructors and selectors. Datatypes may be loose, generated, or free.

### 3.2 Semantic Concepts

The essential semantic concepts for basic specifications are well-known: signatures (of declared symbols), models (interpreting the declared symbols), and sentences (asserting properties of the interpretation), with a satisfaction relation between models and sets of sentences. Defining these (together with some categorical structure, and such that translation of symbols preserves satisfaction) provides a so-called *institution* for basic specifications.

A well-formed basic specification in CASL determines a signature and a set of sentences, and hence the class of all models over that signature which satisfy all the sentences.

*Signatures* $\Sigma = (S, TF, PF, P, \leq)$: A signature $\Sigma$ for a CASL specification consists of a set of sorts $S$, disjoint sets $TF$, $PF$ of total and partial operation symbols (for each profile of argument and result sorts), a set of predicate symbols $P$ (for each profile of argument sorts), and a partial order of subsort embeddings $\leq$. The same symbol may be overloaded, with more than one profile; there are no conditions on the relationship between overloading and subsorts,

and both so-called ad-hoc overloading and subsort overloading are allowed.

*Models* $M \in \mathbf{Mod}(\Sigma)$: A model $M$ provides a non-empty carrier set for each sort in $S$, a partial function for each operation symbol in $PF \cup TF$ (for each of its profiles), a relation for each predicate symbol in $P$ (for each of its profiles), and an embedding for each pair of sorts related by $\leq$. The interpretation of an operation symbol in $TF$ has to be a total function. Moreover, embedding and overloading have to be compatible: embeddings commute with overloaded operations.

*Sentences* $\Phi \in \mathbf{Sen}(\Sigma)$: A sentence $\Phi$ is generally a closed, many-sorted first-order formula. The atomic formulae in it may be equations (strong or existential), definedness and (subsort) membership assertions, and fully-qualified predicate applications. The terms in atomic formulae may be fully-qualified operation applications, variables, explicitly-sorted terms (interpreted as subsort embeddings) or casts (interpreted as projection onto subsorts).

*Satisfaction* $M \vdash \Phi$: The satisfaction of a closed first-order formula $\Phi$ in a model $M$ is as usual regarding quantifiers and logical connectives; it involves the holding of open formulae, and the values of terms, relative to assignments of values to variables.

The value of a term may be undefined when it involves the application of a partial operation symbol (or a cast). When the value of any argument term is undefined, the application of a predicate never holds, and the application of an operation is always undefined (as usual in partial algebra). Definedness of terms also affects the holding of atomic formulae: an existential equation holds when both terms are defined and equal, whereas a strong equation holds when they are both undefined, or defined and equal.

*Sort Generation Constraints* $(S', F') \subseteq (S, F)$, where $F = TF \cup PF$: A sort generation constraint is treated as a further kind of sentence. It is satisfied in a model when the carriers of sorts in $S'$ are generated by functions in $F'$ (possibly from sorts in $S \setminus S'$).

*Institution:* The institution for basic specifications in CASL is given by equipping signatures with morphisms, and models with homomorphisms. A signature morphism $\sigma$ from $\Sigma$ to $\Sigma'$ preserves overloading,

embeddings, and totality of operation symbols. A homomorphism $h : M_1 \to M_2$ ($M_1, M_2 \in \mathbf{Mod}(\Sigma)$) preserves operation values (including definedness), and the holding predicates.

A signature morphism $\sigma$ from $\Sigma$ to $\Sigma'$ determines a translation of sentences from $\mathbf{Sen}(\Sigma)$ to $\mathbf{Sen}(\Sigma')$, and a reduct functor $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$. Translation preserves satisfaction: $M' \vdash \sigma(\Phi)$ iff $\mathbf{Mod}(\sigma)(M') \vdash \Phi$.

*Semantic Functions:* Whereas applications of predicates and operations in the atomic formulae and terms of the CASL *institution* are fully qualified by their profiles, basic specifications in the CASL *language* allow the profiles to be omitted (since they are usually evident from the context). In general, there may be many ways—but possibly none at all—of expanding an atomic formula in CASL by inserting profiles to give a well-sorted fully-qualified atom for constructing a sentence of the underlying institution. The atomic formula is deemed to be well-formed when its expansion is unique (up to the commuting of embeddings with overloaded operations); the axioms of a well-formed basic specification determine a set of sentences of the CASL institution.

In fact the subsorted CASL institution outlined above may be reduced to an ordinary many-sorted CASL institution, by replacing subsort inclusion by explicit embeddings: $\Sigma = (S, TF, PF, P, \leq)$ reduces to $\Sigma^{\#} = (S, TF \cup Emb, PF \cup Proj, P \cup Memb)$ where $Emb = \{emb_{s,s'} \mid s \leq s'\}$ is a set of total operation symbols for subsort embeddings, and the sets $Proj$ (of projections onto subsorts) and $Memb$ (of subsort membership predicates) are defined similarly.

The semantics of a well-formed basic specification in CASL is given by a signature $\Sigma$ together with the class of those models $M \in \mathbf{Mod}(\Sigma)$ that satisfy all the sentences determined by the specification.

### 3.3 Language Constructs

This section provides examples that illustrate the CASL language constructs for use in basic specifications: declarations and definitions (of sorts, operations, predicates, and datatypes), sort generation constraints, and axioms (involving variable declarations, quantifiers, connectives, atomic formulae, and terms). The examples are

shown in *display format*; for input, a suggestive (ASCII or ISO Latin-1) plain text approximation is used, e.g., '$\rightarrow$' is input as '`->`', and '$\forall$' is input as '`forall`'.

Note that CASL allows declarations to be interspersed with definitions and axioms (as illustrated in Section 3.4). Visibility is linear: symbols have to be declared before they can be used (except within datatype declarations, where non-linear visibility allows mutually-recursive datatypes—e.g., *List* and *NeList* on page 10).

*Sorts:* Several sorts may be declared at once, possibly as subsorts of some other sort:[2]

**sorts** $Elem, List$
**sorts** $Nat, Neg < Int$

The values of a subsort may also be defined by a formula, e.g.:

**sort** $Pos = \{n : Nat \bullet n > 0\}$

This corresponds to declaring $Pos < Nat$ and asserting that a value $n$ in $Nat$ is the embedding of some value from $Pos$ iff the formula $n > 0$ holds.

*Operations:* Operations may be declared as total (using '$\rightarrow$') or partial (using '$\rightarrow ?$') and given familiar attributes:

**ops** $0 : Nat$;
$suc : Nat \rightarrow Nat$;
$pre : Nat \rightarrow ? Nat$;
$\_ + \_ : Nat \times Nat \rightarrow Nat, \mathbf{assoc}, \mathbf{comm}, \mathbf{unit}\ 0$

So-called *mixfix notation* is allowed: place-holders for arguments are written as *pairs* of underscores (single underscores are treated as letters in identifiers). All symbols should be input in the ISO Latin-1 character set, but *annotations*[3] may cause them to be displayed differently, e.g., as mathematical symbols.

In simple cases, operations may also be defined at the same time they are declared:

[2] CASL also allows sorts to be declared as isomorphic, with mutual subsort embeddings.
[3] An annotation is an auxiliary part of a specification, for use by tools, and not affecting the semantics of the specification.

**ops** $1 : Nat = suc(0)$;
$dbl(n : Nat) : Nat = n + n$

*Predicates:* Predicate declarations resemble operation declarations, but there is no result sort:

**preds** $odd : Nat$;
$\_\_ < \_\_ : Nat \times Nat$

They too may also be defined at the same time they are declared:

**preds** $even(n : Nat) \Leftrightarrow \neg odd(n)$;
$\_\_ \leq \_\_(m, n : Nat) \Leftrightarrow m < n \vee m = n$

*Datatypes:* A datatype declaration looks like a context-free grammar in a variant of BNF. It declares the symbols on the left of '::=' as sorts, and for each alternative on the right it declares a constructor—possibly with some selectors. When datatypes are declared as 'free', the specified models are as for a free extension in a structured specification: distinct constructor terms of the same sort are interpreted as distinct values, and each declared sort is freely-generated by its constructors.

**type** $Collection ::= empty \mid just(Elem) \mid$
$join(Collection; Collection)$
**free type** $Bit ::= 0 \mid 1$
**free type** $Pair ::= pair(left, right : Elem)$

When there is more than one alternative in a datatype declaration, any selectors are generally *partial* and have to be declared as such, by inserting '?':[4]

**free type** $List ::= nil \mid cons(hd :? Elem; tl :? List)$

Partial selectors can generally be avoided by use of subsort embeddings as constructors:

**free types** $List ::= nil \mid$ **sort** $NeList$;
$NeList ::= cons(hd : Elem; tl : List)$

The last declaration above illustrates non-linear visibility within a list of datatype declarations: *NeList* is used before it has been declared.

[4] Constructors can also be partial.

*Sort Generation Constraints:* The CASL syntax allows datatypes to be declared as 'generated', so that the sorts are constrained to be generated by their constructors (and embedded subsorts):

**generated type** $Bag ::= empty \mid add(Elem; Bag)$
**generated types** $Nat ::= 0 \mid$ **sort** $Pos$;
$Pos ::= suc(pre : Nat)$

More generally, any group of signature declarations can be subject to a sort generation constraint, e.g.:

**generated**
**{ sorts** $Pos < Nat$;
**ops** $0 : Nat$; $suc : Nat \to Pos$ **}**

*Axioms:* Variables for use in axioms may be declared 'globally', in advance:

**vars** $m, n : Nat; p : Pos$
**axioms** $n < m \Rightarrow \ldots; suc(n) = p \Rightarrow \ldots; \ldots$

Variables may also be declared locally to an 'itemized' list of formulae:

**vars** $x, y, z : Elem$
- $x \leq x$
- $x \leq y \wedge y \leq z \Rightarrow x \leq z$

or within a formula using explicit quantification:

$\forall n : Nat \bullet \exists m : Nat \bullet n < m$
$\forall p : Pos \bullet \exists! n : Nat \bullet suc(n) = p$

The logical connectives have their usual interpretation:

$even(n) \Leftrightarrow \neg\ odd(n)$
$m \leq n \Leftrightarrow m < n \vee m = n$
$m < n \Rightarrow \neg\ n = 0$
$even(m + n)\ \mathit{if}\ odd(m) \wedge odd(n)$

*Atomic Formulae:* Definedness assertions can be explicit:

$def\, pre(suc(n)) \wedge \neg def\, pre(0)$

or implicit in existential equations, which are distinguished from strong equations by writing '$\stackrel{e}{=}$' (input as '`=e=`') instead of '=':

$def\, pre(n) \Rightarrow suc(pre(n)) \stackrel{e}{=} pre(suc(n))$
$\neg ok(x, e) \Rightarrow find(x, cons(e, l)) = find(x, l)$

Subsort membership assertions are written suggestively using '$\in$' (input as '`in`'):

$n \in Pos \Leftrightarrow def\, pre(n)$

Applications of predicates are written the same way as those of operations, possibly using mixfix notation.

*Terms:* Constants and variables may be written as sequences of words (optionally separated by *single* underscores—pairs of underscores are reserved for indicating place-holders in mixfix symbols—and decorated with primes) or mathematical signs: *nil*, *empty_set*, $n$, $n'$, $CURRENT\_STATE$.

Applications may be written using standard functional or mixfix notation: $cons(e, l), \{|e|\} \cup s$; they may also be written with explicit qualification, e.g., $pre(n)$ may be written as $(op\ pre : Nat \to ?\ Nat)(n)$. Sorted terms (interpreted as applications of identity or subsort embeddings) are written straightforwardly, e.g.: $dbl(suc(n) : Nat)$. Casts (interpreted as applications of projections onto subsorts, the result of which may be undefined) are written using the reserved word 'as', e.g.: $pre(n\ as\ Pos)$.

### 3.4 Example

The following example illustrates a complete basic specification:

**free types** $Nat ::= 0 \mid$ **sort** $Pos$;
$Pos ::= suc(pre : Nat)$
**op** $pre : Nat \to ?\ Nat$
**axioms**
$\neg def\ pre(0)$;
$\forall n : Nat \bullet pre(suc(n)) = n$

**pred** $even\_\_ : Nat$
**var** $n : Nat$
- $even\ 0$
- $even\ suc(n) \Leftrightarrow \neg even\ n$

Notice that the second line above declares $suc : Nat \to Pos$ and $pre : Pos \to Nat$. The subsequent declaration of $pre : Nat \to? Nat$ allows terms where *pre* is applied to an argument that is of sort *Nat* but not of sort *Pos*—such terms can be perfectly meaningful, e.g., $pre(pre(suc(suc(0))))$.

Further examples may be found in later sections, and in the appendices of the CASL Summary [4].

# 4 Structured Specifications

Structured specifications in CASL are formed in various familiar ways (union, extension, translation, reduction, etc.) starting from basic specifications; they may also be named, to facilitate reuse. Section 4.1 considers various pragmatic issues affecting the CASL design. Section 4.2 presents the main concepts that underly the semantics of structured specifications. Finally, Section 4.3 provides examples that illustrate the CASL language constructs for use in structured specifications.

## 4.1 Pragmatic Issues

*No Model Structure:* The crucial point is that structuring a specification does *not* specify any structure in models! In fact the models of structured specifications are of exactly the same kind as for basic specifications, i.e., algebras interpreting the declared symbols and satisfying all the asserted properties.

For example, consider a specification of the integers. One might choose to structure it as an extension of a specification of natural numbers, rather than giving it as a single basic specification. This choice does not affect the semantics of the specification: neither the signature nor the models reflect the structure of the extension.

Section 5 explains the 'architectural' specifications of CASL, which do allow the structure of models to be specified.

*Names of Symbols:* A general principle underlying the CASL design is '*same name, same thing*'. Thus when one sees two occurrences of the same sort in the same basic specification, one may be sure that they are always interpreted as the same carrier set. For operations

and predicates, the situation is a little more subtle: the 'name' of an operation (or predicate) includes its profile of argument and result sorts, so there need be no relationship at all between say, '+' on integers and '+' on lists or sets, at least in the absence of subsort inclusions.

The 'same name, same thing' principle applies also in unions and extensions—but *not* between named specifications in libraries: the same sort may be used in different named specifications in the same library, with entirely different interpretations; similarly for operations or predicates with the same profiles.

When named specifications are *combined* in the same *structured* specification (by references to their names—perhaps indirectly via other named specifications), any unintended clashes can be eliminated by translating the symbols used in them to new ones. From a methodological point of view, it seems indeed appropriate for the writer of a specification to avoid *accidental* use of the same sort or (qualified) symbol for different purposes, since it could confuse readers. (The same argument does *not* apply to overloading: for example, use of the '$\leq$' predicate for partial orders on different sorts is conventional and nicely emphasises their common properties).

Another point is that in CASL, it is easy to *hide* auxiliary symbols, i.e., symbols that are not inherent to what is to be specified. For example, to specify addition and subtraction on the integers it is common practice to introduce successor and predecessor operations, but they may be regarded as auxiliary and hidden afterwards—they can in any case be recovered using addition and subtraction of 1.

Finally, tools may warn users about multiple declarations of the same name in the same specification, in case they are accidental.

*Generic Extension:* A specification definition *names* a specification, allowing reuse by reference to the name. For example, INT might refer to a specification of the integers. In CASL, a named specification may also have parameters, intended to vary between references; the specification body is an extension of what is specified in the parameters. Each reference to the specification requires instantiation of all its parameters. For example, LIST might refer to a specification that extends a parameter specification named ELEM; any reference to LIST has to provide an argument specification that 'fits' ELEM.

Note that generic extensions in CASL are *not* intended for defining arbitrary functions on specifications, unlike in some other frameworks such as ASL—the CASL user is expected to express the structure of specifications directly using the CASL language constructs that are provided for that purpose.

### 4.2 Semantic Concepts

The notions of signature and class of models are the same as for basic specifications. In fact the structuring part of CASL is (essentially) independent of the details of basic specification: the same structuring may be used regardless of whether basic specifications are restricted (e.g., by eliminating partial functions, subsorts, predicates, or explicit quantifiers) or extended (e.g., to higher-order functions). The reduct homomorphisms induced by signature morphisms in the institution for basic specifications are especially significant in the semantics of structured specifications.

In a specification, the so-called *local environment* records the symbol declarations that are currently visible. For basic specifications, visibility is linear (except within lists of datatype declarations) so the local environment merely grows as one proceeds through the declarations. For structured specifications, however, the local environments at different places may be completely unrelated.

*Semantic Functions:* Structured specifications can have arbitrarily deep structure, and a compositional semantics is appropriate: the denotation of a construct is determined entirely by the denotations of its components. The denotation of a self-contained specification is a signature and class of models for that signature. The denotation of a part that extends a self-contained specification is a (partial) function from signatures to extended signatures, and from corresponding model classes to extended-model classes.

### 4.3 Language Constructs

This section provides examples that illustrate the CASL language constructs for use in structured specifications: translation, reduction, union, extension, free extension, local specifications, specification definitions, generic parameters, instantiation, views, and compound identifiers.

*Translation and Reduction:* Translation of declared symbols to new symbols is specified straightforwardly by giving a list of 'maplets' of the form $old \mapsto new$. Identity maplets $old \mapsto old$ may be abbreviated to *old*, or simply omitted altogether. Optionally, the nature of the symbols concerned (sorts, operations, predicates) may be indicated by inserting the corresponding keywords.

NAT **with** $Nat \mapsto Natural, suc \mapsto succ\_\_$

NAT **with op** $\_ + \_ \mapsto plus$, **pred** $\_\_ < \_ \mapsto lt$

Reduction means removing symbols from the signature of a specification, and removing the corresponding items from models. When a sort is removed, so are all operations and predicates whose profiles include that sort. CASL provides two ways of specifying a reduction: by listing the symbols to be hidden, or by listing those to be left visible, i.e., revealed. In the latter case, (some of) the revealed symbols may also be translated to new symbols.

NAT **hide** $Pos, suc$

NAT **reveal** $Nat, 0, \_\_ + \_\_ , \_\_ < \_ \mapsto lt$

*Unions and Extensions:* The signature of a union of two (or more) specifications is the union of their signatures. The models of a union are those whose reducts to the component signatures are models of the component specifications. There are two extremes: when the specifications have disjoint signatures, their union provides *amalgamation* of their models; when they have the same signature, it provides the *intersection* of the model classes, giving models that satisfy both the specifications at once. For example, the signatures of NAT and STRING might be disjoint, so models of

NAT **and** STRING

would be amalgamations of models of NAT and of STRING, whereas the signatures of MONOID and COMMUTATIVE might be the same, so models of

MONOID **and** COMMUTATIVE

would be those that are simultaneously models of MONOID and of COMMUTATIVE (i.e., commutative monoids).

Extensions may specify new symbols (known as *enrichment*):

> NAT **then**
> **sort** $Nat < Int$;
> **ops** $\_\_ + \_\_ : Int \times Int \to Int$;
> ...

or merely require further properties of old ones:

> COLLECTION **then**
> **axiom** $\forall c : Collection \bullet join(c, c) = c$

An extension is called *conservative* when no models are lost: every model of the specification being extended is a reduct of some model of the extended specification. Whether an extension is conservative or not may be indicated by an annotation (not illustrated here).[5]

*Free Specifications:* The simplest case of a free specification is when the component specification is self-contained. The signature of the specification is unchanged, but the models are restricted to (the isomorphism class of) its initial models. E.g., the only models of the following specification are the standard models of Peano's axioms:

> **free**
> **{ sort** $Nat$; **ops** $0 : Nat$; $suc : Nat \to Nat$ **}**

The conciseness and perspicuity of such specifications may account for the popularity of frameworks that support initiality. When axioms are restricted to positive conditional existential equations, initial models always exist.

More generally, a free specification may be a free extension, e.g.:

> **sort** $Elem$ **then**
> **free**
> **{ type** $Set ::= \emptyset \mid \{|\_\_|\}(Elem) \mid \_\_ \cup \_\_(Set; Set)$
> **op** $\_\_ \cup \_\_ : Set \times Set \to Set$, **assoc**, **comm**, **idem**, **unit** $\emptyset$
> **}**

[5] If conservative extension were to be a construct of the language, its semantics would be the same as for ordinary extension whenever the specified extension happened to be conservative, and otherwise undefined.

Note that free specifications are especially useful for inductively-defined predicates, since only the cases where the predicates hold need be given: all other cases are automatically false. Similarly for partial operations in a free specification, which are as undefined as possible in all its models.

*Local Specifications:* CASL facilitates the hiding of auxiliary symbols by allowing the local scope of their declarations to be indicated. E.g., *suc* below is an auxiliary symbol for use in specifying addition:

**sort** $Nat$
**op** $0 : Nat$
**then local**
**op** $suc : Nat \to Nat$
**within**
**op** $\_\_ + \_\_ : Nat \times Nat \to Nat$
**vars** $m, n : Nat$
- $m + 0 = m$
- $m + suc(n) = suc(m + n)$

Ideally, the operations and predicates of interest are specified directly by their properties, without the introduction of auxiliary symbols that have to be hidden. However, there are classes of models that cannot be specified (finitely) without the use of auxiliary symbols; in other cases the introduction of auxiliary symbols may lead 'merely' to increased conciseness and perspicuity.

*Named Specifications:* Only self-contained specifications can be named—the local environment for a named specification is always empty. Named specifications are intended for inclusion in libraries, see Section 6. Subsequent specifications in the library (or in other libraries) may include a copy of the named specification simply by referring to its name, e.g.:

**spec** NAT = **free { ... }**

**spec** INT = NAT **then free { ... }**

*Generic Parameters:* A parameter is a closed subspecification—typically a reference to a rather simple named specification such as ELEM:

**spec** ELEM = **sort** *Elem*

**spec** LIST [ELEM] =
  **free type** *List* ::= *nil* | *cons*(*Elem*; *List*)

A named specification is essentially an extension of all its parameters. A reference to a named specification with parameters is called an *instantiation*, and has to provide an argument specification for each parameter, indicating how it 'fits' by giving a map from the parameter signature to the argument signature, e.g.:

LIST [NAT **fit** *Elem* ↦ *Nat*]

Since there is only one possible signature morphism from ELEM to NAT, the fitting may be left implicit, and the instantiation written may be written simply as LIST [NAT]. As with translation maps, identity fittings may always be omitted. Of course the map is required to induce not just a signature morphism but also a specification morphism: all models of the argument specification must also be models of the parameter specification.

Sharing between parameter symbols is preserved by fitting, so it may be necessary to rename symbols when separate instantiation of similar parameters is required, e.g.:

**spec** PAIR [**sort** *Elem1*] [**sort** *Elem2*] =
  **free type** *Pair* ::= *pair*(*Elem1*; *Elem2*)

Note that the 'same name, same thing' principle is maintained here. Moreover, to use the same sort name (say *Elem*) in both parameters would require some way of disambiguating the different uses of the name in the body, similar to an explicit renaming.

Sharing of symbols between the body of a generic specification and its arguments in an instantiation is restricted to explicit *imports*, indicated as '**given**':

**spec** LIST_LENGTH [ELEM] **given** NAT =
  **free type** *List* ::= *nil* | *cons*(*Elem*; *List*)
  **op** *length* : *List* → *Nat*

Well-formed instantiations always have a 'push-out' semantics. Had NAT been merely referenced in the body of LIST_LENGTH, an instantiation such as LIST_LENGTH [NAT] would be ill-formed.

*Compound Identifiers:* Suppose that two different instantiations of LIST are combined, e.g.,

LIST [NAT **fit** $Elem \mapsto Nat$] **and**
LIST [CHAR **fit** $Elem \mapsto Char$]

With the previous definition of LIST, an unintentional name clash arises: the sort *List* is declared by both instantiations, but clearly should have different interpretations. To avoid the need for explicit renaming in such circumstances, compound identifiers such as $List[Elem]$ may be used:

**spec** LIST [ELEM] =
  **free type** $List[Elem] ::= nil \mid cons(Elem; List[Elem])$

Now when this LIST is instantiated, the translation induced by the fitting morphism is applied to the component *Elem* also where it occurs in $List[Elem]$, so the sorts in the above instantiations are now distinct: $List[Nat]$ and $List[Char]$.

*Views:* To allow reuse of fitting 'views', specification morphisms (from parameters to arguments) may be themselves named, e.g.:

**spec** PO2NAT : PARTIAL_ORDER **to** NAT =
  $Elem \mapsto Nat, \_\_ \leq \_ \mapsto \_ \leq \_$

The syntax for referencing a named specification morphism, e.g.:

LIST_WITH_ORDER [**view** PO2NAT]

makes it clear that the argument is not merely some named specification with an implicit fitting map, which would be written simply LIST_WITH_ORDER [NAT]

The rules regarding omission of 'evident' maps in explicit fittings apply to named specification morphisms too.

### 4.4 Example

The following example illustrates a complete structured specification definition (referencing a specification named PARTIAL_ORDER, which is assumed to declare the sort *Elem* and the predicate $\_\_ \leq \_\_$):

**spec** LIST_WITH_ORDER [PARTIAL_ORDER] =
  **free type** $List[Elem] ::= nil \mid cons(hd :?Elem; tl :?List[Elem])$
**then**
  **local**
    **op** $insert : Elem \times List[Elem] \to List[Elem]$;
    **vars** $x, y : Elem; l : List[Elem]$
    **axioms**
      $insert(x, nil) = cons(x, nil)$;
      $x \leq y \Rightarrow insert(x, cons(y, l)) = cons(x, insert(y, l))$;
      $\neg(x \leq y) \Rightarrow insert(x, cons(y, l)) = cons(y, insert(x, l))$
  **within**
    **pred** $order[\_ \leq \_] : List[Elem] \times List[Elem]$
    **vars** $x : Elem; l : List[Elem]$
    **axioms**
      $order[\_ \leq \_](nil) = nil$;
      $order[\_ \leq \_](cons(x, l)) = insert(x, order[\_ \leq \_](l))$
**end**

# 5 Architectural Specifications

Architectural specifications in CASL are formed by declaring the *units* that are to be implemented separately, also indicating how they are to be linked together to give the desired result. Section 5.1 considers various pragmatic issues affecting the CASL design. Section 5.2 presents the main concepts that underly the semantics of architectural specifications. Finally, Section 5.3 provides examples that illustrate the CASL language constructs for use in architectural specifications.

## 5.1 Pragmatic Issues

*Reusability:* Whereas the *structuring of specifications* into unions and extensions encourages the reuse of (self-contained) parts of specifications, it does *not* affect the models at all, and the monolithic result of implementing a structured specification is unlikely to be reusable. To allow implementation units to be reusable, CASL provides further constructs for the *specification of structure*.

For a simple example, suppose that one wishes to structure the implementation of LIST [NAT] to include:

– an implementation $N$ of NAT,
– a function $F$ extending *any* such $N$ to an implementation of LIST [NAT], and
– the obvious way of obtaining the desired result: applying $F$ to $N$

The models of the corresponding architectural specification in CASL are required to provide the units $N$ and $F$, as well as their composition. If the implementation $N$ of NAT is subsequently changed, $F$ may be reused, and does not have to be re-implemented. ($F$ may also be changed without changing $N$, of course.)

*Interfaces:* These are the explicit assumptions that parts of an implementation make about other parts. In CASL, interfaces are expressed as ordinary (structured) specifications, asserting that the symbols declared by the specification not only have to be implemented, but also have to satisfy all the asserted properties. A specification of a functional unit involves the specifications of all its arguments and of its result. It is guaranteed that the results of applying functional units to argument units will meet their target specifications, provided that the argument units meet their given specifications.

*Decomposition/Composition:* A crucial aspect of architectural specifications is that they provide decompositions of (possibly large) implementation development tasks into smaller sub-tasks—as well as indicating how to compose (or link together) the results of sub-tasks. A unit specification expresses all that those who are implementing it need to know.

It is clearly desirable to distinguish between structure of specifications and specification of structure, so that specifying (e.g.) INT as an extension of NAT does *not* require separate implementations of these two specifications. What may not be quite so obvious is that the distinction is actually *essential*—at least if one is using the familiar specification structuring constructs provided by CASL. Consider the union of two specifications with some declared common symbols but different axioms: if each specification is implemented separately, without taking account of the properties required by the other specification, it may well happen that the common symbols have different, incompatible implementations which cannot be combined.

### 5.2 Semantic Concepts

A model of an architectural specification consists of:

- a collection of named unit constructors, together with
- the unit (constructor) resulting from a particular composition of those units.

Unit constructors are either constants, or functions from units to units. In the latter case, the functions are always persistent, extending their arguments (the function $F$ considered above should clearly not be allowed to ignore the argument implementation $N$ and incorporate a different implementation of NAT). When functions have more than one argument, the arguments must be compatible, implementing any common symbols in exactly the same way (this follows immediately from the requirement that a function should extend each argument separately).

### 5.3 Language Constructs

This section provides examples that illustrate the CASL language constructs for use in architectural specifications: architectural specification definitions, unit declarations, unit definitions, unit specifications, and unit expressions.

*Architectural Specifications:* A definition of an architectural specification specifies some units and how to compose them, e.g.:

**arch spec** IMP_NAT_LIST =
  **units** $N$ : NAT ;
    $F$ : NAT $\to$ LIST [NAT]
  **result** $F[N]$

A model of the above architectural specification consists of a unit $N$, a unit function $F$, and the unit $F[N]$, which is itself a model of the structured specification LIST [NAT].

*Unit Declarations and Definitions:* A unit declaration names a unit that is to be developed, and gives its type. When the type is an ordinary specification, the unit is to be an ordinary model of it;

when the type is a function type, the unit is to be a function from (compatible) ordinary models to ordinary models that extend the arguments. Some examples of unit declarations:

$N$ : NAT
$L$ : LIST [NAT] **given** $N$
$F$ : NAT $\rightarrow$ LIST [NAT]

The form of unit declaration using '**given**' provides an implicit declaration of a unit function that gets applied just once. If the declaration of $F$ were to be replaced by that of $L$ in the architectural specification example given above (letting the result be simply $L$ as well) then an implementation of the architectural specification would still involve providing a function that could give an implementation of LIST [NAT] extending any implementation $N$ of NAT.

A unit definition names a unit that can be obtained from previous units (in the same architectural specification), possibly involving fitting, hiding, translation, etc.:

$L = F[N]$
$L = F[N$ **fit**$\ldots]$ **hide**$\ldots$

The unit expressions in the right-hand sides of unit definitions are of the same form as used for specifying the result unit of an architectural specification (as explained below).

*Unit Specifications:* A unit specification definition names a unit type, allowing it to be reused. E.g.:

**unit spec** GEN_LIST = NAT $\rightarrow$ LIST [NAT]

A unit declaration may then refer to it, as in $F$ : GEN_LIST.

Architectural specifications themselves may also be used as unit specifications.

*Unit Expressions:* The various forms of unit expression mostly resemble those of structured specifications:

- application (to compatible arguments): $F[N]$, $F[N$ **fit**$\ldots]$
- abstraction: $\lambda N$ : NAT $\bullet$ $\ldots N \ldots$
- translation, reduction: $U$ **with** $\ldots$, $U$ **hide** $\ldots$, $\ldots$
- amalgamation (of compatible units): $N$ **and** $C$

However, the semantics of unit expressions involves operations on individual models, rather than on entire model classes. In particular, amalgamation of models requires that any common symbols are interpreted the same way. Note that abstraction is needed to allow architectural specifications whose results are unit functions.

### 5.4 Example

The following simple example illustrates an architectural specification definition (referencing ordinary specifications named LIST, CHAR, and NAT, assumed to declare the sorts *Elem* and *List*[*Elem*], *Char*, and *Nat*, respectively):

**arch spec** CN_LIST =
  **units**
    $C$ : CHAR ;
    $N$ : NAT ;
    $F$ : ELEM $\rightarrow$ LIST[ELEM]
  **result** $F[C$ **fit** *Elem* $\mapsto$ *Char*$]$ **and** $F[N$ **fit** *Elem* $\mapsto$ *Nat*$]$

Further examples of architectural specifications are given in the CASL Summary [4] and in a recent paper [1].

## 6 Libraries of Specifications

Libraries in CASL are formed by collecting definitions of structured specifications, views, architectural specifications, and unit specifications. Moreover, libraries can refer to specifications defined in other libraries.

Section 6.1 mentions some pragmatic issues affecting the CASL design. Section 6.1 presents the concepts that underly the semantics of libraries. Finally, Section 6.1 briefly illustrates the CASL language constructs for use in libraries.

### 6.1 Pragmatic issues

When specifications are collected into libraries, the question of visibility of symbols between specifications arises. In CASL, the symbols

available in a specification are only those that it declares itself, together with those declared (and not hidden) in named specifications that it explicitly references. Thus when a specification in a library is changed, it is straightforward to locate other specifications that might be affected by the changes.

Another issue concerns visibility of specification names. In CASL, it is linear: a specification may only refer to names of specifications (and views) that precede it in the library. The motivation for this restriction is partly methodological (the library is presented in a bottom-up fashion), partly from implementation considerations (a library can be processed sequentially), and partly from the difficulty of giving a satisfactory formal semantics to mutually-dependent specifications.

CASL provides direct support for establishing distributed libraries on the Internet. A registered library is given a unique name, which is used to refer to it from other libraries when 'downloading' particular specifications. Name servers provide the current locations of registered libraries (before a library is registered, it is referred to by its current URL). Version control is an important pragmatic concern, and the names of CASL libraries incorporate version numbers; however, it is possible to refer to a library without specifying a version (corresponding to using the largest version number that has so far been registered for the library concerned).

It may happen that the same name is used for specifications in different libraries. To avoid confusion between the names of local and downloaded specifications in libraries, a specification that is downloaded from a remote library may be given a different local name. In fact downloading bears a strong resemblance to the FTP command 'get', which provides similar possibilities.

### 6.2 Semantic Concepts

The semantics of a specification in a library is a function of a global environment that maps names of specifications (previously declared or downloaded in the same library) to their denotations. The global environment at the end of the library gives the semantics of the entire library.

A directory of registered libraries maps library names to their registered URLs. Since the names include version numbers, this directory also gives access to previous versions of libraries.

Finally, the semantics of the whole collection of CASL libraries depends on the current state of the Internet, associating URLs to the contents of particular libraries.

### 6.3 Language Constructs

This section outlines some examples that illustrate the CASL language constructs for use in libraries.

*Local Libraries:* The specifications and views in a self-contained library are simply listed in a bottom-up order: a name has to be defined before it is referenced.

**library** COFI_BEANS

**spec** NAT = ...

**view** ... = ...

**arch spec** ... = ...NAT...

**unit spec** ... = ...

*Distributed Libraries:* Other libraries may refer to specifications in registered libraries at other Internet sites by including explicit downloadings, optionally providing a different local name for the remote specification:

**from** COFI_BEANS **get** LIST, NAT $\mapsto$ NATURAL

Libraries may have different versions, indicated by their names, both when defining libraries and downloading specifications from them:

**library** COFI_BEANS **version** 1.2
**from** COFI_PLANT **version** 1.0.1 **get**...

If the version number is omitted in a library definition, it is implicitly 0. The default version when referring to a library is the one that has been registered with the greatest version number.

# 7 Foreground

Having completed the guided tour, let us review the current status of the Common Framework Initiative. Various task groups have been created, concerning language design, semantics, tools, methodology, and reactive systems. There is a substantial amount of interaction between the task groups, which is supported by many of the CoFI participants being active in more than one task group.

The overall coordination of these task groups was managed by the present author from the start of CoFI in September 1995 until August 1998, and subsequently by Don Sannella (Edinburgh). In October 1998, an ESPRIT CoFI Working Group started (CoFI had previously relied on unfunded efforts by its participants, also with initial support from the ESPRIT Compass Working Group until that terminated in March 1996).

*Language Design Task Group:* (Coordination: Bernd Krieg-Brückner, Bremen)

Until October 1998, the main language design task was finalization of the Casl design. The documentation of the final design is given by the Casl Language Summary [4]; a (by now slightly outdated) rationale for the language design was published in 1997 [16]. The semantics, tools, and methodology task groups have all provided essential feedback regarding language design proposals.

The current work in this task group involves the definition of various interesting sublanguages of Casl, e.g., total, many-sorted, equational—mostly corresponding closely to embeddings of the specification languages of other frameworks into Casl [10, 11]. Some extensions are also being investigated, in particular higher-order [7] and state-based specifications (possible extensions for specification of reactive systems is a separate task group).

*Semantics Task Group:* (Coordination: Andrzej Tarlecki, Warsaw)

The major semantics task has of course been defining the formal semantics of Casl, which was produced for earlier versions of the Casl design (in fact Casl had a formal semantics even before its concrete syntax was designed [5]) and which has recently been updated to Casl v1.0 [6].

Apart from consideration of the semantics of sublanguages and extensions of CASL, the main work remaining for the semantics task group is the choice of proof system for CASL.

*Methodology Task Group:* (Coordination: Michel Bidoit, Cachan)

The major task here is the production of a user's guide for CASL. Moreover, various case studies are to be coordinated within this task group. Further topics include the relation of specifications to requirements and code, and a study of software development processes.

*Tools Task Group:* (Coordination: Hélène Kirchner, Nancy)

The concrete syntax for CASL has been designed in joint work between the language design and tools task groups, and its implementation by prototype parsers has provided essential feedback. The various parsers are currently being validated. A LaTeX package for formatting CASL specifications has been developed [17], and can also be used for displaying CASL specifications in HTML.

A major aim of CASL is to support interoperability of existing tools [8, 15, 19]. Collaboration with the developers of tools for other languages will usually be needed to enable the use of CASL specifications in those tools. Other work concerns static analysis and proof tools for CASL, with a prototype (currently for CASL basic specifications only) already implemented using HOL/Isabelle [12].

*Reactive System Task Group:* (Coordination: Egidio Astesiano, Genova)

This task group is mainly concerned with extensions of CASL for reactive system specification, and the combination of CASL with concurrency.

*External Relations Task Group:* (Coordination: the present author)

The design of CASL is based on a (critical) selection of constructs from existing languages, and it should be possible to translate specifications from other languages into (sublanguages or extensions) of CASL. Preliminary investigations have considered ASF+SDF [14] and OBJ3 [9]; moreover, CASL is in many respects close to the KIV language [18]. CoFI does not currently have adequate resources to study and implement translations of other languages into CASL, and

must depend on attracting the interest and collaboration of those who have the necessary expertise.

The design of CASL has been sponsored by IFIP WG1.3 (on Foundations of System Specification), which also provided expert referees to review the proposed design in June 1997 [3]. The ongoing work in CoFI is of great interest to WG1.3, and the author (who was appointed chairman of WG1.3 in 1998) is responsible for liason between CoFI WG and WG1.3.

A major pending task for external relations is to provide an attractive web presentation of CASL. This guided tour of the CASL design, which was presented in half a day as a tutorial at ETAPS'98, should also be improved—suggestions are welcome!—and supplemented by a tutorial with the emphasis on methodology and realistic examples.

*Join now!* All CoFI task groups welcome new participants. Please contact the coordinators via the CoFI web pages [2]. There is a moderated mailing list for each task group, with open subscription, administered by the Majordomo program (majordomo@brics.dk). All CoFI participants are requested to subscribe to a further mailing list, cofi-list@brics.dk (very low-volume, for major announcements only). All CoFI documents are available via the CoFI web pages [2].

*Acknowledgements:* The author is supported by BRICS (Centre for Basic Research in Computer Science), established by the Danish National Research Foundation in collaboration with the Universities of Aarhus and Aalborg, Denmark; by an International Fellowship from SRI International; and by DARPA-ITO through NASA-Ames contract NAS2-98073.

## References

1. Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In *Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *LNCS*, pages 341–357. Springer, 1998.
2. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by WWW[6] and FTP[7].

[6] http://www.brics.dk/Projects/CoFI
[7] ftp://ftp.brics.dk/Projects/CoFI

3. CoFI Language Design Task Group. Response to the Referee Report on CASL. Documents/CASL/RefereeResponse, in [2], August 1997.
4. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [2], October 1998.
5. CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language (version 0.97) – Semantics. Note S-6, in [2], July 1997.
6. CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics (Preliminary Version). Note S-9, in [2], November 1998.
7. Anne Haxthausen, Bernd Krieg-Brückner, and Till Mossakowski. Subsorted partial higher-order logic as an extension of CASL. Note L-10, in [2], October 1998.
8. Einar W. Karlsen. Interoperability of CASL tools using CORBA. Note T-5, in [2], October 1997.
9. Till Mossakowski. Translating other specification languages into CASL. Presented at WADT'98.
10. Till Mossakowski. Sublanguages of CASL. Note L-7, in [2], December 1997.
11. Till Mossakowski. Two "functional programming" sublanguages of CASL. Note L-9, in [2], March 1998.
12. Till Mossakowski, Kolyang, and Bernd Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In *12th Workshop on Algebraic Development Techniques, Tarquinia*, volume 1376 of *LNCS*, pages 333–348. Springer-Verlag, 1998.
13. Peter D. Mosses. CoFI: The Common Framework Initiative for algebraic specification. *Bull. EATCS*, (59):127–132, June 1996.
14. Peter D. Mosses. CASL for ASF+SDF users. In *ASF+SDF'97, 2nd Intl. Workshop on the Theory and Practice of Algebraic Specifications*, volume http://www.springer.co.uk/ewic/workshops/ASFSDF97 of *Electronic Workshops in Computing*. Springer-Verlag, 1997. Invited lecture.
15. Peter D. Mosses. Potential use of SGML for the CASL interchange format. Note T-4, in [2], October 1997.
16. Peter D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 115–137. Springer-Verlag, 1997. Documents/Tentative/Mosses97TAPSOFT, in [2].
17. Peter D. Mosses. Formatting CASL specifications using LaTeX. Note C-2, in [2], June 1998.
18. W. Reif. The KIV-approach to Software Verification. In *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, volume 1009 of *LNCS*, pages 339–368. Springer-Verlag, 1995.
19. Mark van den Brand, Paul Klint, and Pieter Olivier. Aterms: Exchanging data between heterogeneous tools for CASL. Note T-3 (revised draft), in [2], March 1998.

# Recent BRICS Report Series Publications

**RS-98-43** **Peter D. Mosses.** ***CASL: A Guided Tour of its Design*****. December 1998. 31 pp. To appear in Fiadeiro, editor,** ***Recent Trends in Algebraic Development Techniques: 13th Workshop*****, WADT '98 Selected Papers, LNCS, 1999.**

**RS-98-42** **Peter D. Mosses.** ***Semantics, Modularity, and Rewriting Logic*****. December 1998. 20 pp. Appears in Kirchner and Kirchner, editors,** ***International Workshop on Rewriting Logic and its Applications*****, WRLA '98 Proceedings, ENTCS 15, 1998.**

**RS-98-41** **Ulrich Kohlenbach.** ***The Computational Strength of Extensions of Weak König's Lemma*****. December 1998. 23 pp.**

**RS-98-40** **Henrik Reif Andersen, Colin Stirling, and Glynn Winskel.** ***A Compositional Proof System for the Modal $\mu$-Calculus*****. December 1998. 29 pp.**

**RS-98-39** **Daniel Fridlender.** ***An Interpretation of the Fan Theorem in Type Theory*****. December 1998. 15 pp. To appear in** ***International Workshop on Types for Proofs and Programs 1998*****, TYPES '98 Selected Papers, LNCS, 1999.**

**RS-98-38** **Daniel Fridlender and Mia Indrika.** ***An $n$-ary `zipWith` in Haskell*****. December 1998. 12 pp.**

**RS-98-37** **Ivan B. Damgård, Joe Kilian, and Louis Salvail.** ***On the (Im)possibility of Basing Oblivious Transfer and Bit Commitment on Weakened Security Assumptions*****. December 1998. 22 pp. To appear in** ***Advances in Cryptology: International Conference on the Theory and Application of Cryptographic Techniques*****, EUROCRYPT '99 Proceedings, LNCS, 1999.**

**RS-98-36** **Ronald Cramer, Ivan B. Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin.** ***Efficient Multiparty Computations with Dishonest Minority*****. December 1998. 19 pp. To appear in** ***Advances in Cryptology: International Conference on the Theory and Application of Cryptographic Techniques*****, EUROCRYPT '99 Proceedings, LNCS, 1999.**

**RS-98-35** **Olivier Danvy and Zhe Yang.** ***An Operational Investigation of the CPS Hierarchy*****. December 1998.**