BRICS

Basic Research in Computer Science

BRICS RS-97-49 P. D. Mosses: CASL for ASF+SDF Users

CASL for ASF+SDF Users

Peter D. Mosses

BRICS Report Series RS-97-49

ISSN 0909-0878 December 1997

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/97/49/`

CASL for ASF+SDF Users

Peter D. Mosses
pdmosses@brics.dk

Abstract

CASL is an expressive language for the algebraic specification of software requirements, design, and architecture. It has been developed by an open collaborative effort called CoFI (Common Framework Initiative for algebraic specification and development). CASL combines the best features of many previous algebraic specification languages, and it is hoped that it may provide a focus for future research and development in the use of algebraic techniques, as well being attractive for industrial use.

This paper presents CASL for users of the ASF+SDF framework. It shows how familiar constructs of ASF+SDF may be written in CASL, and considers some problems that may arise when translating specifications from ASF+SDF to CASL. It then explains and motivates various CASL constructs that cannot be expressed directly in ASF+SDF. Finally, it discusses the rôle that the ASF+SDF system might play in connection with tool support for CASL.

1 Introduction

CASL is an expressive language for the algebraic specification of software requirements, design, and architecture. It has been developed by an open collaborative effort called CoFI (Common Framework Initiative for algebraic specification and development). This paper presents CASL for users of the ASF+SDF framework.

▷ CASL is intended as the main language of a coherent family of languages.

Vital for the support for CoFI in the algebraic specification community is the coverage of concepts of many existing specification languages. How could this be achieved, without creating a complicated monster of a language? And how to avoid interminable conflicts with those needing a simpler language for use with prototyping and verification tools?

By providing not merely a single CASL language but a coherent language family, CoFI will allow the conflicting demands to be resolved, accommodating advanced as well as simpler languages. At the same time, this family is to be given structure by being organized largely as restrictions and extensions of CASL.

Restrictions of CASL are to correspond to languages used with existing tools for rapid prototyping, verification, term rewriting, etc. Extensions to CASL are to support various programming paradigms, e.g., object-oriented, higher-order, reactive. Apart from such languages, the common framework is also to provide an associated development methodology, training materials, tool support, libraries, a reference manual, formal semantics, and conversion from existing frameworks.

▷ CASL is required to be competitive in expressiveness with various existing languages.

The choice of concepts and constructs for CASL was a matter of finding a suitable balance point between advanced and simpler languages, taking into account its intended applicability: for specifying the functional requirements and design of conventional software packages as abstract data types.

The design of CASL is based on a critical selection of the concepts and constructs found in existing algebraic specification frameworks. The main novelty of CASL lies in its particular *combination* of concepts and constructs, rather than in the latter *per se*. Almost all CASL features may be found (in some form or other) in one or more of the main existing algebraic specification frameworks.

▷ The CASL design has been tentatively approved by IFIP WG 1.3.

The design proposal for CASL [LD97b] was submitted to IFIP Working Group 1.3 (Foundations of System Specification) in May 1997. The proposal provided the abstract syntax of the proposed language, together with an informal summary of the intended well-formedness conditions and semantics [LD97e]; the choice of concrete syntax had not been finalized. Accompanying documents gave the rationale for CoFI [CoF97] and for the Casl design [LD97c], and a draft formal semantics for Casl [Sem97].

The design was tentatively approved at the IFIP WG 1.3 meeting in Tarquinia, June 1997, subject to reconsideration of some particular points [LD97a] and the development of a satisfactory concrete syntax. The abstract syntax and informal summary are currently being finalized [LD97d], after which the formal semantics will be adjusted accordingly. Tools and methodology for Casl are being developed. The concrete syntax of Casl has still not been finalized—that used in the present paper is *tentative*, and subject to change!

▷ CoFI is open to contributions and influence from all those working with algebraic specifications.

The design of Casl has been developed by a varying Language Design task group, coordinated by Bernd Krieg-Brückner, comprising between 10 and 20 active participants representing a broad range of algebraic specification approaches (the CoFI Rationale [CoF97] lists the names of all contributors to CoFI). Numerous study notes have been written on various aspects of language design, and discussed at working and plenary language design meetings. The study notes and various drafts of the design summary were made available electronically and discussed on the associated mailing list (cofi-language@brics.dk).

The openness of the design effort should have removed any suspicion of undue bias towards constructs favoured by some particular 'school' of algebraic specification. It is hoped that Casl incorporates just those features for which there is a wide consensus regarding their appropriateness, and that the common framework will indeed be able to subsume many existing frameworks and be seen as an attractive basis for future development and research—with high potential for strong collaboration.

So much for the background of Casl.

▷ Readers of this paper are assumed to be familiar with the Asf+Sdf language and system.

For an introduction to Asf+Sdf, see [vDHK96].

Plan

First we consider the intersection of ASF+SDF and CASL: those concepts and constructs that are common to both languages. For each such ASF+SDF construct, we see how it might be expressed in CASL, using a tentative concrete syntax.

Then we list the remaining constructs of ASF+SDF: those that cannot (straightforwardly) be expressed in CASL. We motivate the omission of these constructs from CASL.

After that, we list those constructs of CASL that cannot (straightforwardly) be expressed in ASF+SDF, and motivate their inclusion in CASL.

We finish the presentation of CASL by summarizing its constructs, and by giving a few simple examples of CASL specifications. Finally, we consider the rôle that the ASF+SDF system and its users might play in connection with providing tool support for CASL.

▷ All the main points in this paper are displayed like this.

The paragraphs following each point provide details and supplementary explanation. To get a quick overview, simply read the main points and skip the intervening text. It is hoped that the display of the main points does not unduly hinder a continuous reading of the full text. (This style of presentation is borrowed from a book by Alexander [Ale79], where it is used with great effect.)

2 ASF+SDF $\cap$ CASL

▷ ASF+SDF and CASL have a significant number of features in common.

Here we consider the intersection of ASF+SDF and CASL: those concepts and constructs that are common to both languages. For each such ASF+SDF construct, we see how it might be expressed in CASL, using a tentative concrete syntax.

▷ Both frameworks support the basic notions of many-sorted algebra with total functions.

A specification determines a signature (which gives the declared sorts and operation symbols, the latter coming together with the specified argument and result sorts) and the class of those models over the signature that satisfy the specified properties. Each model provides an algebra, i.e., a carrier set for each sort, and a function between carrier sets for each operation symbol.

ASF+SDF and CASL both allow arbitrary overloading, where the same operation symbol can declared with different argument and/or result sorts in the same specification.

In CASL the declaration of sorts $S_1, \ldots, S_n$ is written:

sorts $S_1, \ldots, S_n$

The declaration of a total function symbol f is written:

op $f : S_1 \times \cdots \times S_n \to S$

omitting the arrow for constants:

op $c : S$

▷ ASF+SDF and CASL both allow implicit injections from subsorts.

In CASL the declaration of a subsort S of a sort S' is written:

sort $S < S'$

Overloaded functions are required to commute with subsort injections (is this the case in ASF+SDF?).

▷ ASF+SDF and CASL both allow infix, prefix, postfix, and general mixfix notation, as well as the conventional notation for function application.

In CASL the notation to be used for application is indicated by the use of placeholders '__' in the declared function symbol. E.g., infix notation for applying '+' is specified by declaring the symbol $\_\_+\_\_$. If no placeholders are given, the conventional notation $f(x, y, z)$ is used (omitting the parentheses when applying a constant c).

▷ ASF+SDF and CASL both allow the declarations of sorts, operations, and variables to occur in any order.

In CASL the scope of declarations of sorts, operations, and variables is the entire enclosing list of such items. The declarations are written separated by semicolons, and the keywords introducing subsequent declarations of the same kind may be omitted:

> **sorts** S;
> $\quad\quad S_1 < S$;
> **ops** $f : S_1 \times \cdots \times S_n \to S$;
> $\quad\quad g : S_1 \times \cdots \times S_n \to S$

▷ ASF+SDF and CASL both allow declarations of sorts and operations to be hidden.

In CASL the hiding of sort and operation symbols declared in a specification SP_1 and used in SP_2 is written (tentatively):

> **local** SP_1 **in** SP_2

▷ ASF+SDF and CASL both allow declarations of hidden variables.

In CASL variables are implicitly hidden, their scope always being the enclosing list of declarations and assertions. The declaration of variables $V_1, \ldots, V_n$ of sort S is written:

> **vars** $V_1, \ldots, V_n$: S

▷ ASF+SDF and CASL both allow it to be specified which functions are constructors.

In CASL the constructors (i.e., generators) for some sorts are indicated by grouping them all together in a so-called sort generation constraint:

> **generated sorts** ...; **ops** ... **end**

This constraint eliminates the possibility of "junk" in the carriers for the specified sorts, thereby making structural induction a sound proof principle for them.

▷ ASF+SDF and CASL both allow conditional equations and inequations as axioms.

In CASL conditional equations between terms T, T', T_i, T_i' are written:

$$T_1 = T_1' \wedge \ldots \wedge T_n = T_n' \Rightarrow T = T'$$

or:

$$T = T' \textbf{ if } T_1 = T_1' \wedge \ldots \wedge T_n = T_n'$$

An inequation is written:

$$\neg\ T = T'$$

Axioms are introduced by the keyword **axioms** and separated by semi-colons.

▷ ASF+SDF and CASL both allow specifications to be named and reused.

In CASL a specification named N extending specifications named $N_1, \ldots, N_m$ (corresponding to a module with imports in ASF+SDF) is written (tentatively):

$$\textbf{spec } N = \textbf{enrich } N_1, \ldots, N_m \textbf{ by } SP$$

▷ ASF+SDF and CASL both allow informal comments, and the labelling of individual axioms.

In CASL the concrete syntax for comments and labels has not yet been decided, but it seems likely that end-of-line comments will start with '`%%`', and that labels will be written '`%[...]`'.

▷ ASF+SDF and CASL both support formatting of specifications using mathematical symbols.

In CASL a declared symbol is displayed exactly as input in plain text (ISO Latin-1 character set) unless an explicit display annotation has been given for it. The concrete syntax for display annotations has not yet been decided, but it is expected to allow specification of separate display formats for plain text, LaTeX, HTML, and RTF.

3 Asf+Sdf \ Casl

▷ The features of Asf+Sdf that have been left out of Casl mainly concern lexical syntax and parsing.

These features are primarily of use when specifying the exact concrete syntax of existing languages. This was not regarded as a requirement for a general-purpose specification language for software, since the notation is here chosen by the specifier.

▷ Casl does not allow the specification of lexical syntax rules for literal constants, nor for declared variables.

The lack of lexical syntax in Casl precludes declaring the conventional notation for input of character strings, `"..."` (although display annotations should allow such notation to be used when formatting Casl specifications). Standard numerical notation for integer and real numerical constants can be declared in Casl, albeit somewhat tediously.

In Casl each variable has to be declared explicitly. It is not possible to declare an infinite family of variables of the same form, as one can in Asf+Sdf. Since variables are never exported from a specification (module) in Casl, this omission does not seem to be particularly significant.

▷ Casl does not allow variable declarations to be exported.

This restriction reflects the semantics of Casl specifications (variables are not included in signatures), as well as the conventional treatment of scopes of variables in algebraic specifications.

▷ Casl does not provide built-in list structures.

The list structures provided in Asf+Sdf allow the declaration of operations with arbitrary numbers of arguments. Such operations are particularly useful when specifying the concrete syntax of existing languages, but they do not seem to be needed when specifying software, so they have been excluded from Casl, in the interests of simplicity. Asf+Sdf specifications that exploit list structures will require auxiliary sorts and operations for lists when translated to Casl.

▷ CASL does not allow the specification of priority or associativity for functions.

The CASL concrete syntax (tentatively) adopts a fixed priority scheme for user-declared operations: the infix operations have the lowest priority, then come the prefix operations, and finally the postfix operations. Moreover, infix operations always associate to the left. This would be inadequate for reflecting the concrete syntax of existing languages, but seems to be a reasonable compromise in a software specification language, since it is easy to remember and requires rather few additional parentheses for disambiguation in typical axioms.

▷ CASL does not distinguish functions used for bracketing.

CASL provides ordinary parentheses for use in grouping.

▷ CASL does not provide "otherwise" equations.

Such equations, which cater concisely for default cases, seem closely linked to assumptions of initial semantics and term rewriting implementation for ASF+SDF, which do not apply to CASL.

▷ CASL does not allow cyclic module imports.

This unusual feature of ASF+SDF can indeed be useful, for instance when dividing a context-free grammar for the syntax of a programming language into modules—although the cycles can always be eliminated at the expense of introducing some auxiliary modules. The designers of CASL were unable to give a satisfactory semantics for cyclic references between CASL specifications, due primarily to complications caused by the presence of translation and instantiation—hence the restriction.

▷ CASL could be extended with some of the above features.

Recalling that CASL is intended as the basis for extensions and restrictions, one might consider providing an extension of CASL including the above-mentioned features. One could also provide the restriction of CASL to the features that it shares with ASF+SDF. A combination of extension and restriction of CASL would be needed to provide a language in the CASL family with exactly the features of ASF+SDF.

4 CASL \ ASF+SDF

▷ The features of CASL that are missing from ASF+SDF include explicit treatment of partiality, general first-order axioms, some convenient abbreviations, and constructs for structuring specifications and implementations.

Adding some of these features to ASF+SDF (e.g., first-order axioms) would prohibit rapid-prototyping of specifications using term rewriting; others could probably be added without significant problems.

▷ CASL allows partial functions.

Although total functions are an important special case of partial functions, the latter cannot be avoided in practical applications. CASL adopts the standard mathematical treatment of partiality: functions are 'strict', with the undefinedness of any argument in an application forcing the undefinedness of the result. The lack of non-strict functions seems unproblematic in a pure specification framework, where undefinedness corresponds to the mere lack of a value, rather than to a computational notion of undefinedness. The specification of infinite values such as streams is not supported in CASL, although presumably it will be in some extension.

A partial function declaration is written (tentatively):

$$\mathbf{op}\ f : S_1 \times \cdots \times S_n \rightarrow ?S$$

Partial constants can be declared too:

$$\mathbf{op}\ c :?S$$

▷ CASL provides atomic formulae expressing definedness, as well as both existential and strong equality.

When partial functions are used, the specifier should be careful to take account of the implications of axioms for definedness properties. Thus a clear distinction should be made between *existential* equality, where terms are asserted to have defined and equal values, and *strong* equality, where the terms may also both have undefined values. CASL includes both existential and strong equality, as each has its advantages: existential equality seems most natural to use in conditions of axioms (one does not usually want consequences to follow from the fact that two terms are both undefined), whereas strong equality seems 'safer' to use in unconditional axioms, e.g., when specifying functions inductively.

Tentatively, a strong equation is written:

$$T_1 = T_2$$

and an existential equation as:

$$T_1 \mathrel{=!} T_2$$

Definedness of a term is written:

$$T \ \textbf{defined}$$

▷ CASL allows declarations of predicates.

It is quite common practice to eschew the use of predicates, taking (total) functions with results in some built-in sort of truth-values instead. As with restrictions to conditional equations, this may be convenient for prototyping, but it seems difficult to motivate at the level of using CASL for general specification and verification. Hence predicates may be declared in CASL, tentatively:

$$\textbf{op}\ p : \textbf{pred}(S_1 \times \cdots \times S_n)$$

▷ CASL allows definitions of subsorts, functions, and predicates.

Such definitions abbreviate commonly-occurring combinations of declarations and axioms. A subsort definition is written:

$$\textbf{sort}\ S = \{V : S' \bullet F\}$$

and declares S to be the subsort of S' consisting of just those values of the variable V for which the formula F holds. A total function definition is written:

$$\textbf{op}\ f(V_1 : S_1; \ldots; V_n : S_n) : S = T$$

and a constant definition as:

$$\textbf{op}\ c : S = T$$

CASL also provides similar constructs for defining partial functions and predicates.

▷ CASL allows axioms to be interspersed with declarations.

Why not?

▷ CASL provides concise notation for declaring datatypes with constructors and (optional) selectors.

In a practical specification language, it is important to be able to avoid tedious, repetitive patterns of specification, as these are likely to be carelessly written, and never read closely. The CASL construct of a datatype declaration collects together several such cases into a single abbreviatory construct, which in some respects corresponds to a type definition in STANDARD ML, or to a context-free grammar production in BNF.

A datatype declaration is written (tentatively):

sort $S > A_1 | \cdots | A_n$

It declares a sort, and lists the alternatives A_i for that sort. An alternative may be a constant c, whose declaration is implicit; or it may be a list of sorts, written **sorts** $S_1, \ldots, S_n$, to be embedded as a subsort; or, finally, it may be a 'construct'—essentially an indexed product—written $f(\ldots \times f_i : S_i \times \ldots)$, given by a constructor function f together with its argument sorts S_i, each optionally accompanied by a selector f_i. The declarations of the constructors and selectors, and the assertion of the expected axioms that relate them to each other, are left implicit. When the $>$ above is replaced by $=$, the specified sort is constrained to be generated by the specified constants, embedded subsorts, and constructor functions.

▷ CASL allows all formulae of first-order logic.

In fact many algebraic specification frameworks allow quantifiers and the usual logical connectives: the adjective 'algebraic' refers to the specification of algebras, not to a possible restriction to purely equational specifications, which are algebraic in a different sense.

Universal quantification in CASL is written $\forall V : S \bullet F$. Existential quantification is written using $\exists$, and $\exists_1$ abbreviates a formula expressing existence of a unique value for which F holds.

The standard logical connectives are written $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \Rightarrow F_2$ (alternatively F_2 **if** F_1), $F_1 \Leftrightarrow F_2$, and $\neg F$; the atomic formulae **true** and **false** are provided too.

▷ Casl allows specification of both loose and initial classes of models.

In general, initial models of Casl specifications need not exist, due to the possibility of axioms involving disjunction and negation. When they do exist, the Casl construct for restricting the models of a specification *SP* to the initial ones:

freely *SP*

can be used, ensuring reachability—and also that atomic formulae (equations, definedness assertions, predicate applications) are as false as possible. The latter aspect is particularly convenient when specifying (e.g., transition) relations 'inductively', as it would be tedious to have to specify all the cases when a relation is *not* to hold, as well as those where it should hold.

▷ Casl allows specifications to be combined and extended, and extensions may be required to be free.

For generality, Casl allows specifications with initial semantics to be united with those having loose semantics. This applies also to extensions: the specifications being extended may be either loose or free, and the extending part may be required to be a free extension, which is a natural generalization of the notion of initiality.

Union of specifications in Casl is written (tentatively):

SP_1 **and**... **and** SP_n

and (ordinary) extension as:

enrich *SP* **by** SP'

▷ Casl allows it to be specified that an extension is intended to be conservative.

The case where an extension is 'conservative', not disturbing the models of the specifications being extended, occurs frequently. For example, when specifying a new function on numbers, one does not intend to change the models for numbers. Conservative extension in Casl is written:

enrich conservatively *SP* **by** SP'

▷ CASL allows declared symbols to be translated and/or hidden.

Translation is needed primarily to allow the reuse of specifications with change of notation, which is important since different applications may require the use of different notation for the same entities. But also when specifications that have been developed in parallel are to be combined, some notational changes may be needed for consistency. Translation in CASL is written (tentatively):

SP **renaming** $(\ldots, SY_i \Rightarrow SY'_i, \ldots)$

Hiding symbols ensures that they are not available to the user of the specification, which is appropriate for symbols that denote auxiliary entities, introduced by the specifier merely to facilitate the specification, and not necessarily to be implemented. CASL tentatively provides two constructs for hiding: one where the symbols to be hidden are listed directly (other symbols remaining visible—although hiding a sort entails hiding all function and predicate symbols whose profile involves that sort):

SP **hiding** $(\ldots, SY_i, \ldots)$

the other where only the symbols to be 'revealed' are listed:

SP **revealing** $(\ldots, SY_i, \ldots)$

▷ In CASL the identical declaration of the same symbol in specifications that get combined is regarded as intentional.

Suppose that one unites two specifications that both declare the same symbol: the same sort, or functions or predicates with the same profiles. If this is regarded as well-formed (as it is in CASL) there are potentially (at least) two different interpretations: either the common symbol is regarded as shared, giving rise to a single symbol in the signature of the union, satisfying both the given specifications; or the two symbols are regarded as homonyms, i.e., different entities with the same name, which have somehow to be distinguished in the signature of the union.

CASL (following ASL and LARCH) takes the former interpretation, since the symbols declared by a specification (and not hidden) are assumed to denote entities of interest to the user, and unambiguous notation should be used for them. This treatment also has the advantage of semantic simplicity. However, due to the possibility of unintentional 'clashes' between accidentally-left-unhidden auxiliary symbols, it is envisaged that CASL tools will be able to warn users about such cases. Note that when the two declarations of the symbol arise from the same original specification via separate extensions that later get united, the CASL interpretation gives the intended semantics, and moreover in such cases no warnings need be generated by tools.

▷ CASL allows generic specifications, with instantiation affecting compound identifiers.

The parameters SP_i of a generic specification, which is written:

$$\textbf{spec } N[\ldots,SP_i,\ldots] = SP$$

are simply dummy parts of the specification (declarations of symbols, axioms) that are intended to be replaced systematically whenever the name N of the generic specification is referred to in an instantiation, which is written:

$$N[\ldots,SP'_i,\ldots] \textbf{ where } (\ldots, SY_j \Rightarrow SY'_j, \ldots)$$

The classic example is the generic specification of lists of arbitrary items: the parameter specification merely declares the sort of items, which gets replaced by particular sorts (e.g., of integers, characters) when instantiated. For a generic specification of *ordered* lists, the parameter specification would also declare a binary relation on items, and perhaps insist that it have (at least) the properties of a partial order.

It is possible to view generic specifications as a particular kind of loose specification, with instantiation having the effect of tightening up the specification. Thus generic lists of items are simply lists where the items have been left (extremely) loosely specified. Instantiating items to integers then amounts to translating the entire specification of lists accordingly (so that e.g. the first argument of the 'cons' function is now declared to be an integer rather than an item) and forming its union with the specification of integers—the CASL treatment of common symbols in unions dealing correctly with the two declarations of the sort of integers.

▷ CASL allows the use of compound identifiers for symbols in generic specifications.

The observant reader may have noticed that in the example described above, two different instantiations of the generic lists (say, for integers and characters) would declare the same sort symbol for the two different types of lists, causing problems when these get united. CASL allows the use of compound sort identifiers of the form:

$SY[\ldots,SY_i,\ldots]$

in generic specifications; e.g., the sort of lists may be a symbol formed with the sort of items as a component, say *List*[*Elem*]. The translation of the parameter sort to the argument sort affects this compound sort symbol for lists too, giving distinct symbols such as *List*[*Int*], *List*[*Char*]for lists of integers and lists of characters, and thereby avoiding the danger of unintended identifications and the need for explicit renaming when combining instantiations.

▷ CASL provides architectural specifications, for specifying the structure of models (which is generally not affected by the structure of the specification).

The structuring constructs considered above allow a large specification to be presented in small, logically-organized parts, with the pragmatic benefits of comprehensibility and reusability. In CASL, the use of these constructs has absolutely no consequences for the structure of models, i.e., of the code that implements the specification. For instance, one may specify integers as an extension of natural numbers, or specify both together in a single basic specification; the models are the same.

It is especially important to bear this in mind in connection with generic specifications. The definition of a generic specification of lists of arbitrary items, and its instantiation on integers, does *not* imply that the implementation has to provide a parametrized program module for generic lists: all that is required is to provide lists of integers (although the implementor is free to *choose* to use a parametrized module, of course). Sannella, Sokołowski, and Tarlecki [SST92] provide extensive further discussion of these issues.

In contrast, an architectural specification requires that any model should consist of a collection of separate component units that can be composed in a particular way to give a resulting unit. Each component unit is to be implemented separately, providing a decomposition of the implementation task into separate subtasks with clear interfaces.

In CASL, an architectural specification consists of a collection of component unit specifications, together with a description of how the implemented units are to be composed. A model of such a specification consists of a model for each component unit specification, and the described composition.

▷ CASL allows libraries to be distributed across sites on the Internet.

An ordered collection of named specifications forms a library in CASL. Linear visibility is assumed: a specification in a library may refer only to the specifications that precede it.

▷ Libraries may be located at particular sites on the Internet, and their current contents referenced by means of URL's.

Given that there will be more than one CASL library of specifications (at least one library per project, plus one or more libraries of standard CASL specifications) the issue of how to refer from one library to another arises. The standard WWW notion of a Uniform Resource Locator (URL) seems well-suited for this purpose: a library may be identified with some index file located in a particular directory at a particular site, accessible by some specified protocol (e.g., FTP).

▷ A library may require the 'down-loading' of particular named specifications from other libraries each time it is used.

Rather than allowing individual references to names throughout specifications to include the URLs of the relevant libraries (which might be inconvenient to maintain when libraries get reorganized), CASL provides a separate construct for down-loading named specifications from another library. Optionally, the specification may be given a local name different from its original name, so that one may easily avoid name clashes; the resemblance of this construct to the familiar FTP command 'get' is intentional.

5 CASL Overview and Examples

This section gives a concise overview of all the main CASL features, covering both that are in common with ASF+SDF as well as those that are not.

▷ Basic specifications in CASL list declarations, definitions, and axioms.

Functions are partial or total, and predicates are allowed. Subsorts are interpreted as embeddings. Axioms are first-order formulae built from definedness assertions and both strong and existential equations. Sort generation constraints can be applied to groups of declarations. Datatype declarations are provided for concise specification of enumerations, unions, and products.

▷ Structured specifications in CASL allow translation, reduction, union, and extension of specifications.

Extensions may be required to be conservative and/or free; initiality constraints are a special case. A simple form of generic specification is provided, together with instantiation involving parameter-fitting translations that affect compound identifiers.

▷ Architectural specifications in CASL express implementation structure.

The specified software is to be composed from separately-developed, reusable units with clear interfaces.

▷ Libraries in CASL provide collections of named specifications.

Downloading involves retrieval of specifications from distributed libraries.

ASF+SDF **Examples in** CASL

spec Bool_Example =
  sort BOOL;
  ops true, false : BOOL;
    __|__ , __&__ : BOOL × BOOL → BOOL;
    not : BOOL → BOOL;
  var *Bool* : BOOL;
  axioms
  %% disjunction
    %[B1] true | *Bool* = true;
    %[B1] false | *Bool* = *Bool*;
  %% etc.

spec Another_Bool_Example =
  sorts BOOL = true | false |
    __|__(BOOL×BOOL) | __&__(BOOL×BOOL) |
    not(BOOL);
  ops __|__ , __&__ : BOOL × BOOL → BOOL;
    __|__ , __&__ : **assoc**, **comm**, **idem**;
    __|__ : **unit** false;
    __&__ : **unit** true;
  var *Bool* : BOOL;
  axioms
  %% etc.

generic spec Lists[**sort** ITEM] =
  sorts LIST[ITEM]; ITEM < LIST[ITEM];
  ops nil : LIST[ITEM];
    __ ⋈ __ : LIST[ITEM]×LIST[ITEM]→LIST[ITEM];
    __ ⋈ __ : **assoc**, **unit** nil;
  vars i : ITEM; l_1 l_2, l_3 : LIST[ITEM];
  axioms $l_1 \bowtie i \bowtie l_2 \bowtie i \bowtie l_3 = l_1 \bowtie i \bowtie l_2 \bowtie l_3$

6 ASF+SDF Support for CASL?

The Common Framework Initiative has already benefited from the participation of ASF+SDF users at CoFI meetings and on the CoFI mailing lists during the design of CASL. It is not expected (nor even desirable) that ASF+SDF users should suddenly switch to being CASL users. They could, however, help substantially with the future development of CASL in connection with the following points:

▷ ASF+SDF could be used for implementing the proposed concrete syntax for CASL, and for checking the parsing of example CASL specifications.

ASF+SDF is especially well-suited for rapid prototyping of the CASL concrete syntax, and for checking not only that example specifications conform to that syntax, but also that they are unambiguous.[1]

▷ ASF+SDF could be used for automating translation between CASL and other languages, also in connection with the interoperability of tools that were originally developed for use with different languages.

In particular, translation from (a suitable sub-language of) CASL to ASF+SDF would give CASL users access to the rewriting capabilities of ASF+SDF.

▷ The techniques developed for formatting ASF+SDF specifications could be applied to CASL.

Display annotations in CASL are for determining the formatting of function symbols in applications, but they give little direct control of layout. The ASF+SDF experience with pretty-printing documents via an intermediate box language seems attractive.

▷ Some CASL constructs could perhaps be incorporated in the design of future versions of ASF+SDF.

This list is not intended to be exhaustive! The author would like to hear of further ideas for ASF+SDF support for CASL.

[1]Straight after the ASF+SDF Workshop, Mark van den Brand did this for CASL basic specifications, and it should be straightforward to extend his ASF+SDF grammar to generate parsers for structured and architectural specifications.

References

[Ale79] Christopher Alexander. *A Timeless Way of Building.* Oxford University Press, 1979.

[CoF97] CoFI. CoFI – The Common Framework Initiative for Algebraic Specification and Development – Rationale. CoFI Document: Rationale. WWW[2], FTP[3], May 1997.

[LD97a] CoFI Task Group on Language Design. Response to the Referee Report on Casl. CoFI Document: CASL/RefereeResponse. WWW[4], FTP[5], August 1997.

[LD97b] CoFI Task Group on Language Design. Casl – The CoFI Algebraic Specification Language – Design Proposal. CoFI Document: CASL/Proposal. WWW[6], FTP[7], May 1997.

[LD97c] CoFI Task Group on Language Design. Casl – The CoFI Algebraic Specification Language – Rationale. CoFI Document: CASL/Rationale. WWW[8], FTP[9], May 1997.

[LD97d] CoFI Task Group on Language Design. Casl – The CoFI Algebraic Specification Language – Summary. CoFI Document: CASL/Summary. WWW[10], FTP[11], May 1997.

[LD97e] CoFI Task Group on Language Design. Casl – The CoFI Algebraic Specification Language – Summary, version 0.97. CoFI Document: CASL/Summary-v0.97. WWW[12], FTP[13], May 1997.

[2] http://www.brics.dk/Projects/CoFI/Documents/Rationale/
[3] ftp://ftp.brics.dk/Projects/CoFI/Documents/Rationale/
[4] http://www.brics.dk/Projects/CoFI/Documents/CASL/RefereeResponse/
[5] ftp://ftp.brics.dk/Projects/CoFI/Documents/CASL/RefereeResponse/
[6] http://www.brics.dk/Projects/CoFI/Documents/CASL/Proposal/
[7] ftp://ftp.brics.dk/Projects/CoFI/Documents/CASL/Proposal/
[8] http://www.brics.dk/Projects/CoFI/Documents/CASL/Rationale/
[9] ftp://ftp.brics.dk/Projects/CoFI/Documents/CASL/Rationale/
[10] http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/
[11] ftp://ftp.brics.dk/Projects/CoFI/Documents/CASL/Summary/
[12] http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary-v0.97/
[13] ftp://ftp.brics.dk/Projects/CoFI/Documents/CASL/Summary-v0.97/

[Sem97] CoFI Task Group on Semantics. CASL – The CoFI Algebraic Specification Language (version 0.97) – Semantics. CoFI Note: S-6. WWW[14], FTP[15], July 1997.

[SST92] Don Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

[vDHK96] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping. An Algebraic Specification Approach.* World Scientific, Singapore, 1996.

Addendum:

This paper also appears in *ASF+SDF'97, Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshops in Computing*, http://www.springer.co.uk/ewic/workshops/ASFSDF97. Springer-Verlag, 1997.

[14]http://www.brics.dk/Projects/CoFI/Notes/S-6/

[15]ftp://ftp.brics.dk/Projects/CoFI/Notes/S-6/

Recent BRICS Report Series Publications

RS-97-49 Peter D. Mosses. *CASL for ASF+SDF Users*. December 1997. 22 pp. Appears in *ASF+SDF'97, Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshops in Computing*, http://www.springer.co.uk/ewic/workshops/ASFSDF97. Springer-Verlag, 1997.

RS-97-48 Peter D. Mosses. *CoFI: The Common Framework Initiative for Algebraic Specification and Development*. December 1997. 24 pp. Appears in Bidoit and Dauchet, editors, *Theory and Practice of Software Development. 7th International Joint Conference CAAP/FASE*, TAPSOFT '97 Proceedings, LNCS 1214, 1997, pages 115–137.

RS-97-47 Anders B. Sandholm and Michael I. Schwartzbach. *Distributed Safety Controllers for Web Services*. December 1997. 20 pp. To appear in *European Theory and Practice of Software. 1st Joint Conference FoSSaCS/FASE/ESOP/CC/TACAS*, ETAPS '97 Proceedings, LNCS, 1998.

RS-97-46 Olivier Danvy and Kristoffer H. Rose. *Higher-Order Rewriting and Partial Evaluation*. December 1997. 20 pp. Extended version of paper to appear in *Rewriting Techniques and Applications: 9th International Conference*, RTA '98 Proceedings, LNCS, 1998.

RS-97-45 Uwe Nestmann. *What Is a 'Good' Encoding of Guarded Choice?* December 1997. 28 pp. Revised and slightly extended version of a paper published in *5th International Workshop on Expressiveness in Concurrency*, EXPRESS '97 Proceedings, volume 7 of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers.

RS-97-44 Gudmund Skovbjerg Frandsen. *On the Density of Normal Bases in Finite Field*. December 1997. 14 pp.

RS-97-43 Vincent Balat and Olivier Danvy. *Strong Normalization by Run-Time Code Generation*. December 1997.

RS-97-42 Ulrich Kohlenbach. *On the No-Counterexample Interpretation*. December 1997. 26 pp.