CoFI: The Common Framework Initiative for Algebraic Specification and Development

Peter D. Mosses*

BRICS,** Dept. of Computer Science, University of Aarhus
Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark

Abstract. An open collaborative effort has been initiated: to design a common framework for algebraic specification and development of software. The rationale behind this initiative is that the lack of such a common framework greatly hinders the dissemination and application of research results in algebraic specification. In particular, the proliferation of specification languages, some differing in only quite minor ways from each other, is a considerable obstacle for the use of algebraic methods in industrial contexts, making it difficult to exploit standard examples, case studies and training material. A common framework with widespread acceptance throughout the research community is urgently needed.
The aim is to base the common framework as much as possible on a critical selection of features that have already been explored in various contexts. The common framework will provide a family of specification languages at different levels: a central, reasonably expressive language, called CASL, for specifying (requirements, design, and architecture of) conventional software; restrictions of CASL to simpler languages, for use primarily in connection with prototyping and verification tools; and extensions of CASL, oriented towards particular programming paradigms, such as reactive systems and object-based systems. It should also be possible to embed many existing algebraic specification languages in members of the CASL family.
A tentative design for CASL has already been proposed. Task groups are studying its formal semantics, tool support, methodology, and other aspects, in preparation for the finalization of the design.

1 Background

A large number of algebraic specification frameworks have been provided during the past 25 years of research, development, and applications in this area.

Table 1 lists the main frameworks, with a rough indication of their chronology. Some of them are ambitious, wide-spectrum frameworks, equipped with a full

* E-mail: pdmosses@brics.dk

** Centre for Basic Research in Computer Science, The Danish National Research Foundation

ABEL ETL SPECTRAL SPECTRUM LPG	
EXTENDED-ML OBSCURE TROLL	*1990's*
ACT-TWO ASF+SDF RSL	
ASL LARCH RAP SMoLCS	
ASSPEGIQUE COLD-K	*1980's*
ACT-ONE PLUSS	
CIP OBJ	
CLEAR	*1970's*
LOOK	

Table 1. Algebraic specification frameworks

software development methodology; others are much more modest, consisting essentially of a prototyping or verification tool and its associated language. For references and further details, see the COMPASS bibliography [3] and *Recent Trends in Data Type Specification* [6].

No de-facto standard framework for algebraic specification has emerged.

Although some of the existing frameworks are relatively popular, with substantial communities of users, none has achieved such widespread support as for example that enjoyed by VDM and Z in the model-oriented specification community. (The fact that VDM and Z have a lot of minor dialects is beside the point.) Most algebraic frameworks were developed at particular university departments, or by international collaboration between individual researchers, and each framework tends to be used rather locally. The main exceptions are LARCH and OBJ; one might mention here also ACT-ONE/TWO, RSL, and SPECTRUM. Not surprisingly, it seems that most frameworks strongly reflect the convictions held by their originators, which tends to make them less acceptable to those holding different convictions.

The lack of a common, widely-supported framework for algebraic specification is a major problem.

In particular, it is an obstacle for the adoption of algebraic methods for use in industrial contexts, and makes it difficult to exploit standard examples, case studies and educational material. But even within academia, the diversity of explanations of basic algebraic specification notions in text-books, and the lack of a common corpus of accepted examples, form a significant hindrance to dissemination. And the various tools that have been developed for prototyping, verifying, and otherwise supporting the use of algebraic specifications, are each generally available only in connection with just one framework. Moreover, the prospects for continued support and development of locally-developed frameworks are usually quite uncertain, which discourages their adoption by industry and investment in training in their use.

It is time to agree on the fundamental concepts and constructs that could form the basis of a common framework.

The various groups working on algebraic specification frameworks have already had ample opportunity to develop and experiment with their own particular variations on the theme of algebraic specification. A substantial collective experience and expertise in the design and use of such frameworks has been accumulated. If we cannot agree *now* on what are the *essential* concepts and constructs, there would seem to be little grounds for belief that such agreement could ever be achieved.

This paper presents CoFI*: The Common Framework Initiative for algebraic specification and development, explains the (tentative) design of* CASL*: The* CoFI *Algebraic Specification Language, and sketches plans for the future.*

The author is currently the overall coordinator of CoFI. It should be emphasized that the ideas presented below stem from a voluntary international collaboration involving many participants (see the Acknowledgements at the end), and it would be both difficult and inappropriate to accredit particular ideas to individuals.

By the way: CoFI is intended to be pronounced like 'coffee', and CASL like 'castle'.

All the main points in this paper are summarized like this.

The paragraphs following each point provide details and supplementary explanation. To get a quick overview of CoFI and CASL, 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 [1], where it is used with great effect.)

2 CoFI

The initial idea for a common framework initiative was conceived in June 1994, by members of COMPASS and IFIP WG 1.3.

COMPASS (1989–96) was an ESPRIT Basic Research WG (3264, 6112) involving the vast majority of the European sites working on algebraic specification [7]. IFIP WG 1.3 (Foundations of System Specification) was founded in 1992 (originally with the number 14.3) and has members not only from the major European sites but also from other continents.

In fact the idea of developing a common algebraic specification framework had been suggested for inclusion in the original COMPASS WG proposal in 1988—but subsequently dropped, as it was considered unlikely to be achievable. By 1994, however, the area had matured sufficiently to encourage reconsideration of the idea of a common framework.

By September 1995 the main aims had been clarified, and CoFI*: The Common Framework Initiative started.*

A joint meeting of COMPASS and IFIP WG 1.3 at Soria Moria, near Oslo, in September 1995 decided to set up the Common Framework Initiative, and various task groups were formed. Since the termination of COMPASS in April 1996, IFIP WG 1.3 has taken the sole responsibility for the future of the initiative, and for approving any proposals that it might make.

The overall aims of CoFI [8] are:

- A common framework for algebraic specification and software development is to be designed, developed, and disseminated.
- The production of the common framework is to be a collaborative effort, involving a large number of experts (30–50) from many different groups (20–30) working on algebraic specifications.
- In the short term (e.g., by 1997) the common framework is to become accepted as an appropriate basis for a significant proportion of the research and development in algebraic specification.
- Specifications in the common framework are to have a uniform, user-friendly syntax and straightforward semantics.
- The common framework is to be able to replace many existing algebraic specification frameworks.
- The common framework is to be supported by concise reference manuals, users' guides, libraries of specifications, tools, and educational materials.
- In the longer term, the common framework is to be made attractive for use in industrial contexts.
- The common framework is to be available free of charge, both to academic institutions and to industrial companies. It is to be protected against appropriation.

The common framework is to allow and be useful for:

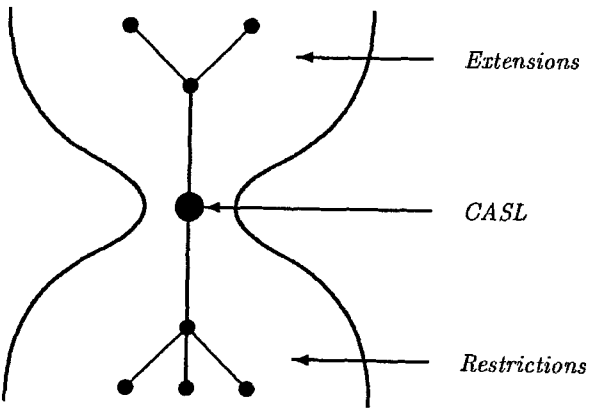
- Algebraic specification of the functional requirements of software systems, for some significant class of software systems.
- Formal development of design specifications from requirements specifications, using some particular methods.
- Documenting the relation between informal statements of requirements and formal specifications.
- Verification of correctness of development steps from (formal) requirements to design specifications.
- Documenting the relation between design specifications and implementations in software.
- Exploration of the (logical) consequences of specifications: e.g., rewriting, theorem-proving, prototyping.
- Reuse of parts of specifications.
- Adjustment of specifications and developments to changes in requirements.
- Providing a library of useful specification modules.
- Providing a workbench of tools supporting the above.

In effect, the above list is the requirements specification for the common framework, avoiding premature design decisions. It provided the starting-point for the actual design of the common framework.

An early but key design decision was that the common framework should provide a coherent family of languages, all extensions or restrictions of some main algebraic specification language.

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 language but a coherent language family, CoFI allows the conflicting demands to be resolved, accommodating advanced as well as simpler languages. At the same time, this family is given a clear structure by being organized as restrictions and extensions of a main language, which is to be the main topic of the documentation (reference manual, user's guide, text book) and strongly identified with the common framework.



The main language of the common framework family is required to be competitive in expressiveness with various existing languages.

The choice of concepts and constructs for the main language was a matter of finding a suitable balance point between the advanced and simpler languages. It was decided that its intended applicability should be for specifying the functional requirements and design of conventional software packages as abstract data types.

Restrictions of the main language are to correspond to languages used with existing tools for rapid prototyping, verification, term rewriting, etc.

These may be syntactic and/or semantic restrictions. The restricted languages need not have a common kernel—although presumably all restrictions will allow at least unstructured, single- or many-sorted equational specifications.

Existing tools typically restrict the use of sorts and overloading, allow only a restricted class of axioms, and may require specifications to be 'flattened'.

The semantics of a specification in a restricted language may be inherited from the semantics of the main language, although some simplifications should usually be possible.

Extensions to the main language are to support various programming paradigms, e.g., object-oriented, higher-order, reactive.

These are to be obtained from the main language (or perhaps from mildly restricted languages) by syntactic and/or semantic extensions. The extended languages need not have a common super-language, and indeed, there may be technical difficulties in combining various extensions.

The semantics ascribed to a specification in the main language by an extension is required to be essentially the same as its original semantics.

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.

A framework is more than just a language! Many existing algebraic specification frameworks have not had sufficient resources to develop all the required auxiliary documents, which has severely hampered their dissemination. By pooling resources in CoFI, this problem may be avoided.

Regarding tools, the aim is to make it possible to exploit existing tools in connection with the common framework, using an interchange format [2].

One of the attractions of having a common framework is to facilitate building up a library of useful specifications in a single language. Libraries of specifications have previously been proposed, but the variety of languages involved was always a problem.

Conversion from existing frameworks is vital, not only to be able to reuse existing specifications, but also to encourage users to migrate from their current favourite framework to the common framework.

The tentative design of the main CoFI *Algebraic Specification Language, called* CASL, *was completed in December 1996, and is currently undergoing closer investigation by task groups concerned with issues of language design, methodology, semantics, and tool support.*

It was felt that CoFI participants had sufficient collective expertise and experience of designing algebraic specification frameworks, and knowledge of existing frameworks, to allow the rapid development of a tentative design for CASL by selecting and combining familiar concepts and constructs. (In fact it turned out

that collaborative design of a language was a good way of *forcing* the participants to understand each other's views in depth—more reliably than through the attendance of presentations at conferences.) But then it was felt essential to allow time for a closer study before finalizing the design, in case any infelicities had crept in. In particular, it should be checked that there are no inherent semantic problems with the chosen combination of constructs.

Some CoFI task group meetings are to be held just before this paper is presented at TAPSOFT'97. On the basis of the investigations made by these groups, a definite complete proposal for the design of CASL will be submitted to IFIP WG 1.3 for approval at its meeting in June 1997.

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

The tentative design of CASL was 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. Numerous study notes were written on various aspects of language design, and discussed at working and plenary language design meetings. The study notes and various drafts of the tentative design summary were made available electronically and comments solicited via the associated mailing list (cofi-language@brics.dk).

This 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.

All the CoFI task groups welcome new active participants. See the descriptions of the task groups on the CoFI WWW pages [9], and contact the coordinators of the task groups directly.

3 CASL

This section presents the main points of the tentative design of CASL.

> *The tentative 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*. All CASL features may be found (in some form or other) in one or more of the main existing algebraic specification frameworks, with a couple of minor exceptions: with subsorts, it was preferred to avoid the (non-modular) condition of 'regularity'; and with libraries, it was felt necessary to cater for links to remote sites.

The aim with CASL *is to provide an expressive specification language with simple semantics and good pragmatics.*

The reader may notice below that from a theoretical point of view, some CASL constructs could be eliminated, the same effect being obtainable by combined use of the remaining constructs. This is because CASL is not intended as a general kernel language with constructs that directly reflect theoretical foundations, and where one would need to rely on 'syntactic sugar' to provide conciseness and practicality. By including abbreviatory constructs in the syntax of CASL, their uniformity with the rest of the syntax may be enforced, and in any case they add no significant complications at all to the CASL semantics.

CASL *is for specifying requirements and design of conventional software packages.*

All CASL constructs are motivated by their usefulness in general algebraic specification: there are no special-purpose constructs, only for use in special applications, nor is CASL biased towards particular programming paradigms.

The tentative design of CASL *provides the abstract syntax, together with an informal summary of the intended well-formedness conditions and semantics; the choice of concrete syntax has not yet been made.*

It is well-known that people can have strong feelings about issues of concrete syntax, and it was felt necessary to delay all discussions of such issues until after the tentative design of the CASL abstract syntax and its intended semantics had been decided. Consequently, CASL is at the time of writing without any concrete syntax at all, which makes it difficult to give accurate illustrative examples of specifications.

Let us consider the concepts and constructs of so-called basic specifications in CASL, *followed by structured specifications, architectural specifications, and finally libraries of specifications.*

First, here is a concise overview of the complete language. *Basic specifications* in CASL denote classes of partial first-order structures: algebras where the functions are partial or total, and where also 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 stated. *Structured specifications* allow translation, reduction, union, and extension of specifications. Extensions may be required to be persistent and/or free; initiality constraints are a special case. Type definitions are provided for concise specification of enumerations and products. A simple form of generic (parametrized) specifications is provided, together with instantiation involving parameter-fitting translations. *Architectural specifications* express that the specified software is to be composed from separately-developed, reusable units with clear interfaces. Finally, *libraries* allow the (distributed) storage and retrieval of named specifications.

The remarks below explain how CASL caters for the various features, and attempts to justify the tentative design choices that have been made. The complete tentative abstract syntax of CASL is given in an appendix. For a systematic presentation of the intended semantics of CASL constructs, see the CASL Tentative Design Summary [4], available for browsing on WWW via the CoFI Home Page [9].

3.1 Basic Specifications

Partiality

Functions may be partial, the value of a function application in a term being possibly undefined. Total functions may be declared as such.

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 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 language.

Signatures of CASL specifications distinguish between partial and total functions, the latter being required to be interpreted in all models as partial functions that happen to be totally-defined. It should be straightforward to define restricted languages that correspond to the conventional partial and total algebraic specification frameworks.

Atomic formulae expressing definedness are provided, 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. The tentative design of 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.

Definedness of a term could be expressed by an existential equality, at the expense of writing the same term twice. It was deemed important to be able to express definedness of the value of a term directly by an atomic formula.

The underlying logic is 2-valued.

Just because the values of terms may be undefined, one need not let this affect formulae (although various other frameworks have chosen to do so). In CASL, a (closed) formula is either satisfied or not, in any particular model. This keeps the interpretation of the logical connectives completely standard, and avoids a range of questions for which there do not appear to be any optimal solutions.

Subsorts and Overloading

Functions (and predicates) may be overloaded, the same symbol being declared for more than one sequence of argument sorts. Argument sorts are related by subsort inclusions, but no 'regularity' conditions are imposed on declarations.

Here, the design of CASL found itself in a dilemma: it was recognized as highly desirable to provide support for the concept of subsorts and overloading (e.g., to allow the specification of natural numbers as a subsort of the integers, with the usual functions on natural numbers being extended to integers), but the notion of 'regularity' of signatures, as adopted in order-sorted algebras [5], was found to have some drawbacks. Finally, it was decided to put no conditions at all on the declarations of overloaded functions, but instead to require that any uses of overloaded functions in terms should be sufficiently disambiguated, ensuring that different parses of the same term (involving different overloadings) always have the same semantics. The consequences for parsing efficiency of this tentative decision are currently being investigated.

Subsort inclusions are represented by embedding functions, whose insertion in terms may be left implicit. The corresponding inverse projection functions from supersorts to subsorts are partial.

In order-sorted algebra, subsort inclusions are modelled as actual set-theoretic inclusions between the corresponding carriers, whereas in CASL, they are more general, being arbitrary embeddings. This extra generality allows one to specify e.g. that integers are to be a subsort of the approximate real numbers, without requiring all models to use the same representation of each integer as for the corresponding approximate real.

Thanks to the possibility of partial functions in CASL, the projection functions from supersorts to subsorts can be given a straightforward algebraic semantics.

Predicative sort definitions allow the concise specification of subsorts that are determined by the values for which particular formulae hold.

It was realized, during the design of subsorting in CASL, that one may distinguish two different uses of subsorts: (i) in the extension of a subalgebra, e.g., from natural numbers to integers, and (ii) to indicate the domain of definition of a partial function, e.g., the even numbers for integer division by 2. In (i) the values of the subsort(s) are generated implicitly by the declarations of operations of

the subalgebra, whereas in (ii) it may be more convenient to characterize them explicitly by some predicate or formula. To cater for the latter, CASL provides a construct called a predicative sort definition. This declares a new sort consisting of those values of another sort for which a particular formula holds—this might be written $\{x : s \mid P[x]\}$, where $P[x]$ is some formula involving the variable x ranging over the sort s. (More precisely, the values of the new sort are the projections of values of sort s.)

Formulae

The usual first-order quantification and logical connectives are provided.

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. But of course many prototyping systems do restrict specifications to (conditional) equations, so as to be able to use term rewriting techniques in tools; this will be reflected in restrictions of CASL to sublanguages.

Predicates for use in atomic formulae may be declared.

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, and combined using the standard logical connectives.

Sort Generation Constraints

It may be specified that a sort is generated by a set of functions, so that proof by induction is sound for that sort.

For generality, CASL does not restrict all models to be finitely-generated (i.e., reachable). The specifier may indicate that a particular sort (or set of sorts) is to be generated by a particular set of functions, much as in LARCH.

3.2 Structured Specifications

A structured specification is formed by combining specifications in various ways, starting from basic specifications. The structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style. (Specification of the *architecture* of models in CASL is addressed in the next section.)

Translation and Hiding

The symbols declared by a specification may be translated to different ones, and they may be 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.

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), the other where only the symbols to be 'revealed' are listed.

Union and Extension

Specifications of independent items may be combined, and subsequently extended with specification of further sorts, functions, predicates, and/or properties.

The most fundamental way of combining two independent specifications is to take their union. Models of the united specification have to provide interpretations of all the symbols from the two specifications. The provision of union allows independent parts of a specification to be presented separately, thereby increasing the likelihood that they will be reusable in various contexts. CASL provides a construct for taking the union of any number of specifications.

Extension of a specification allows the addition of further functions (and predicates) on already-specified sorts, perhaps adding new sorts as well. It is also possible with extension to add further properties, either concerning already-specified symbols or ones being introduced in the extension itself. The CASL construct for extension allows arbitrary further bits of structured specification to be added to the union of any number of specifications. In fact union itself is essentially just an empty extension.

It may be declared whether or not the models of the specifications being extended are to be preserved.

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. For generality, CASL allows the specifier to indicate for each of the extended specifications whether its models are intended to be preserved or not.

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.

Initiality and Freeness

Specifications generally have loose semantics: all models of the declared symbols that enjoy the specified properties are allowed. However, it may also be specified that only initial models of the specification are allowed.

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 models to the initial ones 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.

Specifications with loose and initial semantics may 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.

Type Definition Groups

A type definition group allows the concise declaration of one or more sorts together with constructor and selector functions, with some implicit axioms relating the constructors and 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 type definition group collects together several such cases into a single abbreviatory construct, which in many respects corresponds to a type definition in STANDARD ML, or to a context-free grammar in BNF.

A type definition group consists of one or more type definitions (possibly together with some axioms). Each type definition declares a sort, and lists the alternatives for that sort. An alternative may be a constant, whose declaration is implicit; or it may be a sort, to be embedded as a subsort (of the sort of the type definition); or, finally, it may be a construct—essentially a product—given by a constructor function together with its argument sorts, each optionally accompanied by a selector. The declarations of the constructors and selectors, and the assertion of the expected axioms that relate them to each other, are implicit.

Special cases of type definitions are enumerations of constants (although no ordering relation or successor function is provided) and unions of subsorts. Notice that we now have three distinct ways of specifying subsorts: directly, or by predicative sort definitions, or by type definitions. (One may also represent a subsort as a unary predicate, although then it cannot be used in declarations of function or predicate symbols, nor when declaring the sorts of variables.)

The semantics of a type definition group involves free extension.

The intended semantics is that the only values of the sorts declared by a type definition group are those that can be expressed using the listed constants, subsort embeddings, and constructor functions. Moreover, different constants or constructors of the same sort are supposed to have distinct values: there should be no 'confusion'. Such properties could (at least in the absence of user-specified axioms) be spelled out using sort-generation constraints and first-order axioms, but in fact the intended semantics is precisely captured by the notion of initial semantics (or, in the case that alternatives involve sorts declared outside the type-definition group, free extension).

A type definition group may be used as an item of a basic specification.

A type definition group is essentially something like a complete basic specification, and can be combined with other specifications in structured specifications. But especially when specifying 'small' type definitions, e.g., enumerations of constants or unions of subsorts, it would often be awkward to have to separate this part and make an explicit extension of it. Thus CASL allows a type definition group to be used directly as an item of a basic specification, with semantics corresponding to the introduction of an implicit extension.

Naming and Generics

A (possibly-structured) specification may be given a name; subsequent references to the name are equivalent to writing out the specification again.

The naming of a specification in CASL serves two main purposes (apart from the purely informal one of suggesting the intentions of the specifier!): to avoid the verbatim repetition of the same specification part within one specification; and to allow its insertion in a library of specifications, so that the specification may be reused simply by referring to its name in all subsequent specifications.

A specification may be made generic, by declaring some parameters which are to be instantiated with 'fitting' arguments whenever reference to the name of the specification is made.

The parameters of a generic specification are simply dummy parts of the specification (declarations of symbols, axioms) that are intended to be replaced systematically whenever the name of the generic specification is referred to. 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.

Note that, in contrast to some other specification languages, the parameter here is *not* a bound variable, whose occurrences in the body (if any) should be replaced by the argument specification. Such a λ-calculus form of parametrization would allow the specifier to introduce quite general functions from specifications to specifications; in CASL, the intention is that one always uses the constructs described in this section directly when combining specifications. Moreover, the usefulness of specification functions that ignore their parameter(s) is questionable; with the CASL form of generics, the parameter is automatically extended by the generic specification.

A generic specification may have several parameters. Any common symbols have to be instantiated the same way (the situation is analogous to an extension, where common symbols declared by the specifications that are being extended are regarded as identical). Thus if a generic specification is to have two independent parameters, say pairs of two (possibly) different sorts of items, one has to use different symbols for the two sorts. Although this seems to be a coherent design, CASL does differ in its treatment of parameters from that found in many previous specification languages, so a careful explanation of this point will have to be provided in the supporting manuals and guides.

The semantics of instantiation of generic specifications corresponds to a push-out construction.

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.

In fact the semantics of instantiation in CASL corresponds closely to the above explanation. Under suitable conditions, it corresponds to a push-out construction on specifications.

> *The use of compound identifiers for symbols in generic specifications allows the symbols declared by instantiations to depend on the symbols provided by the argument specifications.*

The observant reader may have noticed that in the example given 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 in generic specifications; e.g., the sort of lists may be a symbol formed with the sort of items as a component. The translation of the parameter sort to the argument sort affects this compound sort symbol for lists too, giving distinct symbols for lists of integers and lists of characters, thereby avoiding the danger of unintended identifications and the need for explicit renaming when combining instantiations.

3.3 Architectural Specifications

> *The structure of a specification does not require models to have any corresponding structure.*

The structuring constructs considered in the preceding section 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 [10] 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.

A unit may be required to provide an extension of other units that are being implemented separately. The compatibility of implementations of any common declared symbols in the extended units has to be ensured.

In general, the individual units may be regarded as functions: they correspond to parametrized program modules that extend their arguments. For example, one may specify a unit that is to extend any implementation of integers with an implementation of lists of integers, thus separating the task of implementing integers as a self-contained sub-task, and with the implementation of lists being allowed to apply the specified functions and predicates on integers. The specification of a unit consists of the specification of each argument that is to be extended, and the specification of the extension itself. These argument and result specifications form the interfaces of the unit.

A unit implementing lists of integers is not allowed to replace the implementation of integers by a different one! The argument has to be preserved, i.e., the unit has to be a persistent function. To cater for this, the result *signature* of each unit has to include each argument *signature*—any desired hiding has to be left to when units are composed. Since each symbol in the union of the argument signatures has to be implemented the same way in the result as in each argument where it occurs, the arguments must already have the same implementation of all common symbols. In CASL, this is built into the semantics of architectural specifications, and the specifier does not have to spell out the intended identity between parts of arguments, nor between arguments and results (in contrast to a previous approach to architectural specifications [10]). The description of the composition of units is only well-formed when it ensures that units with potentially-incompatible implementations of the same symbols cannot be combined as arguments.

When the resulting unit is composed, the symbols defined by a unit may be translated or hidden.

In the example considered above, one may alternatively specify a more general unit that it is to extend any implementation of arbitrary items (not just implementations of integers) with lists. Such a unit can then be applied to an implementation of integers, the required fitting of items to integers being described as part of the composition of units.

Architectural specifications and the specifications of their components may be named, and subsequently referenced.

Although architectural and component specifications have different semantics and usage compared to structured specifications, there is a similar need to be able to name them and reuse them by simply referring to their names.

3.4 Libraries of Specifications

Named specifications of various kinds can be collected in libraries.

As indicated above, CASL allows specifications to be named. 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. In fact the possibility of allowing cyclic references in CASL libraries (as in ASF+SDF) was considered, but in the presence of translation and instantiation, it seemed that the semantics would not be sufficiently straightforward.

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. However, a named specification at a remote library may well refer to other named specifications in that library (or in other libraries) and it would be unreasonable to require explicit mention of such auxiliary specifications, so these get down-loaded implicitly, with special local names that cannot clash with ordinary names.

The overall effect is that one may use a down-loading construct to provide access to named specifications located at remote libraries, without having to worry about anything but the names of the required specifications and the URL of the library. Notice that no construct is provided for down-loading an entire library: the names of the specifications required have to be listed. This ensures that references to names can always be checked for local declaration, before down-loading occurs.

4 Foreground

This section sketches the plans for the immediate future of the Common Framework Initiative. Up-to-date information may be found via the CoFI WWW pages [9].

The tentative design of CASL *will be revised, if necessary, on the basis of its investigation by the various* CoFI *task groups.*

The main responsibility here is on the Semantics task group, which is currently making a critical review of the informal explanation of the intended semantics in the existing CASL language summary, and contemplating what semantic entities would be needed for a formal semantics. This should reveal any ambiguities and incompletenesses in the informal explanation, as well as providing grounds for belief in the existence of a reasonable semantic model for the combined CASL constructs.

Other task groups are active as well: the Language Design task group is to test the tentative CASL design by expressing standard examples in CASL—it is also considering the issue of restrictions and extensions of CASL, for instance to check that a higher-order extension could be provided without undue difficulty; the Methodology task group is considering the development of implementations from CASL specifications; and the Tools task group is working on the issue of interfacing CASL with existing specification languages and tools, as well as clarifying what basic tools for CASL will need to be implemented.

The revised design, together with proposals for concrete syntax and tool support, will be submitted to a meeting of IFIP WG 1.3 in June 1997.

Any problems with the tentative CASL design should have been discovered and rectified before the revised design proposal is submitted. It is hoped that several alternative proposals for concrete syntax, with illustrative examples, will have been made by then; whether it will be so easy to reach agreement on just one proposal is perhaps not so clear at present.

A lot of work remains to be done...

The approval of a CASL design will be just the start of the main CoFI work: progressing from ideas to their realization in documentation, methodology, and tools. Although CoFI has already come quite a long way on the basis of voluntary effort and local support at various sites, and the expected redirection of future development towards languages and tools based on CASL should provide further resources, international funding for CoFI will be needed to allow the realization of its full potential for industrial applications.

Acknowledgements

The following (45) individuals have contributed to the common framework initiative by commenting on various CoFI documents or attending CoFI meetings: Egidio Astesiano, Hubert Baumeister, Jan Bergstra, Gilles Bernot, Didier Bert, Mohammed Bettaz, Michel Bidoit, Pietro Cenciarelli, Maria Victoria Cengarle, Maura Cerioli, Christine Choppy, Ole-Johan Dahl, Hans-Dieter Ehrich, Hartmut Ehrig, Jose Fiadeiro, Marie-Claude Gaudel, Chris George, Joseph Goguen, Radu Grosu, Anne Haxthausen, Jim Horning, Hélène Kirchner, Hans-Jörg Kreowski, Bernd Krieg-Brückner, Pierre Lescanne, Tom Maibaum, Grant Malcolm, Karl Meinke, Till Mossakowski, Peter D. Mosses, Peter Padawitz, Fernando Orejas, Olaf Owe, Gianna Reggio, Horst Reichel, Gerard Renardel, Erik Saaman, Don Sannella, Giuseppe Scollo, Amilcar Sernadas, Andrzej Tarlecki, Eelco Visser, Eric Wagner, Michał Walicki, and Martin Wirsing. (Apologies to anyone who has been inadvertently omitted.)

Groups at the following sites have generously hosted CoFI meetings (1995–97): Aarhus, Bremen, Edinburgh, Munich (LMU), Munich (TUM), Oslo, Oxford, Paris (LIENS/ENS), Paris (LSV/ENS de Cachan). Some CoFI meetings were much facilitated by support from COMPASS.

References

1. C. Alexander. *A Timeless Way of Building.* Oxford University Press, 1979.
2. M. Bidoit, C. Choppy, and F. Voisin. Interchange format for inter-operability of tools and translation. In Haveraaen et al. [6], pages 102–124.
3. M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Software Development*, volume 501 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.
4. CoFI. CASL: The CoFI algebraic specification language, tentative design: Language summary. Notes Series NS-96-15, BRICS, Department of Computer Science, University of Aarhus, 1996.
5. J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, Computer Science Lab., SRI International, 1989.
6. M. Haveraaen, O. Owe, and O.-J. Dahl, editors. *Recent Trends in Data Type Specification*, volume 1130 of *Lecture Notes in Computer Science.* Springer-Verlag, 1996.
7. B. Krieg-Brückner. Seven years of COMPASS. In Haveraaen et al. [6], pages 1–13.
8. P. D. Mosses. CoFI: The common framework initiative for algebraic specification. *Bulletin of the EATCS*, June 1996.
9. P. D. Mosses, editor. *CoFI: Common Framework Initiative for Algebraic Specification*, URL: http://www.brics.dk/Projects/CoFI/, 1997.
10. D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Inf.*, 29:689–736, 1992.

Appendix: Tentative Abstract Syntax of CASL

The abstract syntax is presented as a set of production rules in which each entity is defined in terms of its constituent parts. The productions form a context-free grammar. The notation X*, X+, X? indicates the repetition of X any number of times, at least once, and at most once, respectively.

The order in which components of constructs are currently listed does not necessarily correspond to that to be used in the concrete representation.

Identifiers

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

Basic Specifications

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

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

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

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

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

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

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

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

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

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

Basic Specifications with Subsorts

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

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

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

Structured Specifications

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

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

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

Generic Specifications

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

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

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

Architectural Specifications

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

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

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

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

Specification Libraries

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