CASL: The Common Algebraic Specification Language [1]

Egidio Astesiano [a], Michel Bidoit [b], Hélène Kirchner [c], Bernd Krieg-Brückner [d], Peter D. Mosses [e], Donald Sannella [f] and Andrzej Tarlecki [g]

[a] *Dipartimento di Informatica e Scienze dell'Informazione, Universitá di Genova, Italy*

[b] *LSV, CNRS and École Normale Supérieure de Cachan, France*

[c] *LORIA-CNRS, Nancy, France*

[d] *Bremen Institute of Safe Systems, Department of Computer Science, Universität Bremen, Germany*

[e] *BRICS and Department of Computer Science, University of Aarhus, Denmark*

[f] *Laboratory for Foundations of Computer Science University of Edinburgh, UK*

[g] *Institute of Informatics, Warsaw University and Institute of Computer Science, PAS, Warsaw, Poland*

Abstract

CASL is an expressive language for the formal specification of functional requirements and modular design of software. It has been designed by CoFI, the international *Common Framework Initiative for algebraic specification and development.* It is based on a critical selection of features that have already been explored in various contexts, including subsorts, partial functions, first-order logic, and structured and architectural specifications. CASL should facilitate interoperability of many existing algebraic prototyping and verification tools.

This paper gives an overview of the CASL design. The major issues that had to be resolved in the design process are indicated, and all the main concepts and constructs of CASL are briefly explained and illustrated—the reader is referred to the CASL Language Summary for further details. Some familiarity with the fundamental concepts of algebraic specification would be advantageous.

[1] This research was supported by the ESPRIT-funded CoFI Working Group 29432.
[2] Corresponding Author: Bernd Krieg-Brückner

1 Background

Algebraic specification is one of the most extensively-developed approaches in the formal methods area. The most fundamental assumption underlying algebraic specification is that programs are modelled as *many-sorted algebras* consisting of a collection of sets of data values together with functions over those sets. This level of abstraction is commensurate with the view that the correctness of the input/output behaviour of a program takes precedence over all its other properties. Another common element is that specifications of programs consist mainly of logical *axioms*, usually in a logical system in which equality has a prominent role, describing the properties that the functions are required to satisfy—often just by their interrelationship. This *property-oriented* approach is in contrast to so-called *model-oriented* specifications in frameworks like VDM [36] which consist of a simple realization of the required behaviour. However, the theoretical basis of algebraic specification is largely in terms of constructions on algebraic models, so it is at the same time much more model-oriented than approaches such as those based on type theory (see e.g. [58]), where the emphasis is almost entirely on syntax and formal systems of rules, and semantic models are absent or regarded as of secondary importance.

The past 25 years has seen a great deal of research on the theory and practice of algebraic specification. Overviews of this material include [5,9,13,41,68,69,76]. Developments on the foundational side have been balanced by work on applications, but despite a number of success stories, industrial adoption has so far been limited. The current proliferation of *algebraic specification languages* is seen as a significant obstacle to the dissemination and use of these techniques. Despite extensive past collaboration between the main research groups involved and a high degree of agreement concerning the basic concepts, the field has given the appearance of being extremely fragmented, with no *de facto* standard specification language, let alone an international standard. Moreover, although many *tools* supporting the use of algebraic techniques have been developed in the academic community, none of them has gained wide acceptance, at least partly because each tool uses a different specification language.

The dozens of algebraic specification languages that have been developed all support the basic idea of using axioms to specify algebras, but differ in design choices concerning syntax (concrete and abstract) and semantics.

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

The *Common Framework Initiative for algebraic specification and development*, CoFI [18], started in September 1995 [40,53]. The aims and scope were formulated as follows.

The *aims* of CoFI are to provide a common framework:

- by a collaborative effort
- for algebraic specification and development
- attractive to researchers as well as for use in industry
- providing a common specification language with uniform, user-friendly syntax and straightforward semantics
- able to subsume many previous frameworks
- with good documentation and tool support
- free—but protected (cf. GNU [28])

The *scope* of CoFI is:

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

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

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

The Casl design effort started in September 1995, as a common effort of the COMPASS Working Group [40] and IFIP WG1.3 (Foundations of System Specification). An initial design was proposed [19] in May 1997 (with a language summary, abstract syntax, formal semantics, but no agreed concrete syntax) and tentatively approved by IFIP WG1.3. The report of the IFIP referees [26] on the initial Casl design proposal suggested reconsideration of several points in the language design, and requested some improvements to the documents describing the design; the response by the language designers to the referees [20] indicates the improvements that were made in the revised language design and its documentation. Apart from a few details, the design was finalized in April 1998, with a complete draft language summary available, including concrete syntax. CASL version 1.0 [21] was released in October 1998; the formal semantics given for the proposed design has also been updated to reflect the changes [24].

CASL subsumes many previous languages for the formal specification of functional requirements and modular software design. Tools for CASL are interoperable, i.e. capable of being used in combination rather than in isolation. CASL interfaces to existing tools extend this interoperability.

Even though the intention was to base the design of CASL on a critical selection of concepts and constructs from existing specification languages, it was not easy to reach a consensus on a coherent language design. A great deal of careful consideration was given to the effect that the constructs available in the language would have on such aspects as the methodology and tools. A complete formal semantics for CASL was produced in parallel with the later stages of the language design (in fact CASL had a formal semantics even before its concrete syntax was designed [23]), and the desire for a relatively straightforward semantics was one factor in the choice between various alternatives in the design.

2 Overview

CASL represents a consolidation of past work on the design of algebraic specification languages. With a few minor exceptions, all its features are present in some form in other languages but there is no language that comes close to subsuming it. Designing a language with this particular novel collection of features required solutions to a number of subtle problems in the interaction between features.

It was clear from the start that no single language could suit all purposes. On the one hand, sophisticated features are required to deal with specific programming paradigms and special applications. On the other hand, important methods for prototyping and reasoning about specifications only work in the *absence* of certain features: for instance, term rewriting requires specifications with equational or conditional equational axioms.

CASL is therefore the heart of a *family* of languages. Some tools will make use of well-delineated *sub-languages* of CASL obtained by syntactic or semantic restrictions [49], while *extensions* of CASL are being defined to support various paradigms and applications. The design of CASL took account of some of the planned extensions, particularly one that involves higher-order functions [50], and this had an important impact on decisions concerning details of abstract syntax.

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

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

The above division of Casl into parts is orthogonal to taking sublanguages of Casl. The Casl language design integrates several different *aspects*, which are here explained separately:

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

In the sequel, each part of Casl is presented in turn, in a kind of guided tour, considering all the aspects listed above before proceeding to the next part.

3 Basic Specifications

A Casl basic specification denotes a class of Casl *models*, which are *many-sorted partial first-order structures*: algebras where the functions are partial or total, and where also predicates are allowed. These are classified by *signatures*, which list sort names, partial and total function names, and predicate names, together with profiles of functions and predicates. The sorts are pre-ordered by a subsorting relation, which is interpreted in models using embeddings (rather than set-theoretic inclusions) required to commute with overloaded functions.

A Casl basic specification includes *declarations*, to introduce components of signatures, and *axioms*, to give properties of those structures that are to be considered as the *models*[3] of the specification. Axioms are written in first-order logic (potentially using quantifiers and all the usual logical connectives) built over atomic formulae which include strong and existential equalities,

[3] We inherit from the theory of institutions the usual somewhat ambiguous use of the term 'model': either as an arbitrary Casl model over a given signature, or as a model of a specification. When the ambiguity may be dangerous, however, we use the term 'structure' in the former case.

definedness formulae, and predicate applications; generation constraints are allowed too.

The interpretation of formulae is as in classical two-valued first-order logic, in contrast to some other frameworks that accommodate partial functions, e.g., VDM [36]. Concise syntax is provided for subsort, operation, and predicate definitions, and for specifications of 'datatypes' with constructor and selector functions.

3.1 Pragmatic Issues

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

For instance, consider the familiar operations on (possibly-empty) lists: the list constructor *cons* would be declared as total, whereas the list *hd* and *tl* selectors could be partial, being undefined on the empty list.

free type *List* ::= *nil* | *cons*(*hd* :?*Elem*; *tl* :?*List*)

Alternatively, the domain of definition of the selectors may be made explicit by introducing the subsort *NeList* of non-empty lists, and declaring the *hd* and *tl* selectors to be total functions on that subsort (more on subsorts below).

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

In the presence of partiality, equations may require definedness: so-called 'existential' equations require it, 'strong' equations do not.[4] In general, it is appropriate to use existential equations in conditions (since properties do not usually follow from undefinedness) but strong equations when defining partial

[4] An existential equation between two terms of the same sort holds when both terms are defined and equal; a strong equation holds additionally when they are both undefined.

functions inductively. So CASL allows both kinds of equations.

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

Logic and Predicates: CASL is based on classical two-valued first-order logic. It supports user-declared predicates, which have some advantages over the (total) Boolean functions that were used instead of predicates in most previous algebraic specification languages. For example, predicates hold minimally in initial models. This allows to specify only positively where a predicate holds and to omit the negative cases, which are automatically determined by initial (or free) semantics. When any argument of a predicate is undefined, the predicate application never holds.

CASL provides the standard universal and existential quantification and logical connectives, as in ordinary first-order predicate logic. The motivation for this departure from the most traditional algebraic approaches is expressiveness: restricting to conditional equations sometimes requires quite contrived specifications. For instance, it is a straightforward exercise to specify when a string is a permutation of another using quantifiers, and negation provides the complementary property; but the latter is quite awkward to specify using only (positive) conditional equations.

Equational and conditional equational specification frameworks are however provided as sublanguages of CASL, simply by restricting the use of quantifiers and logical connectives [45].

Classes of Models: CASL adopts so-called loose semantics for basic specifications: all structures satisfying the axioms are taken as models of a basic specification. This is appropriate for its intended use as a requirements specification language, where the class of models (i.e., potential implementations) should be as large as possible, so as to leave the implementor room for design decisions and to avoid overspecification. It is also possible in CASL to specify the restriction of models to the class of generated models (only expressible values are included, hence no 'junk' data are allowed and properties may be proved by induction) or to the class of initial or free models (providing minimal satisfaction of atomic formulae, thus in particular preventing 'confusion' between data). Of course, neither generated nor initial/free models need exist if arbitrary first-order axioms are used—the class of models may even be empty.[5]

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

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

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

Casl does not impose any conditions of 'regularity', 'coherence', or 'sensibleness' on the relationship between overloading and subsorts [14]. This is partly for simplicity (no such conditions are required for the semantics of Casl), partly because most such conditions lack modularity (which is a disadvantage in connection with structured specifications). Note that overloaded constants are allowed in Casl (e.g., *empty* may be declared to be a constant of various sorts of collections).

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

[5] Of course, specifications in purely equational frameworks may also have empty model classes, in the presence of hierarchical or data constraints.

(only models generated by the constructors are taken, but the same data may be constructed in different ways), or free (only models where the declared sorts are freely generated by the constructors are taken, which captures the standard datatypes found in programming languages; cf. **free type** above).

3.2 Semantic Concepts

The essential semantic concepts for basic specifications are well-known: signatures (of declared symbols), models (interpreting the declared symbols), and sentences (asserting properties of the interpretation), with a satisfaction relation between models and sets of sentences. Defining these (together with some categorical structure, and such that translation of symbols preserves satisfaction) provides a so-called *institution* [30]. A well-formed basic specification in CASL determines a signature and a set of sentences, and hence the class of all models over that signature which satisfy all the sentences.

Signatures: $\Sigma = (S, TF, PF, P, \leq)$: A signature Σ for a CASL specification consists of a set of sorts S, disjoint sets TF, PF of total and partial operation symbols (for each profile of argument and result sorts), a set of predicate symbols P (for each profile of argument sorts), and a subsorting pre-order [6] $\leq$ on the set S of sorts. The same symbol may be overloaded, with more than one profile; there are no restricting conditions on the relationship between overloading and subsorts (as in some other languages such as OBJ, cf. [31,32]), and both so-called ad-hoc overloading and subsort overloading are allowed, cf. [14].

Models: $M \in \mathbf{Mod}(\Sigma)$: A Σ-*model* M provides:

- a non-empty carrier set for each sort in S,
- a total function for each operation symbol in TF (for each of its profiles),
- a partial function for each operation symbol in PF (for each of its profiles),
- a relation for each predicate symbol in P (for each of its profiles), and
- an embedding for each pair of sorts related by $\leq$.

Embeddings are arbitrary (total) injections; composition of subsort embeddings yields a subsort embedding, and the embedding from any sort to itself is the identity. They also determine partial projections (from supersorts to subsorts) and subsort membership predicates (that hold on those values of a supersort that are in the image of the subsort embedding). Moreover, embedding and overloading have to be compatible: embeddings commute with

[6] That is: a reflexive and transitive relation.

overloaded operations. See [24] (and also [14,49]) for the rather obvious formal statement of these requirements.

The categorical structure of Σ-models is given by the expected notion of homomorphism. A homomorphism $h : M_1 \to M_2$ between models $M_1, M_2 \in \mathbf{Mod}(\Sigma)$ is a total function between their carriers that preserves values of all operations (including subsort embeddings) and respects predicates (so that if a predicate holds for some data in M_1 then it holds for the values of h on these data in M_2).

Sentences: $\Phi \in \mathbf{Sen}(\Sigma)$: A Σ-sentence Φ is generally a closed first-order formula. The atomic formulae in it may be equations (strong or existential), definedness and (subsort) membership assertions, and predicate applications. The terms in atomic formulae may be variables, applications of operations to terms of the sorts determined by the profile of the operation, explicitly-sorted terms (interpreted using subsort embeddings) or casts to a subsort of a term of a supersort (interpreted as projection onto the subsort).

Satisfaction: $M \models \Phi$: The satisfaction of a closed first-order Σ-formula Φ in a Σ-model M is defined as usual regarding quantifiers and logical connectives; it involves the holding of open formulae, and the values of terms, relative to assignments of values to variables. The value of a term may be undefined when the application of a partial operation symbol (or a cast) occurs in it. When the value of any argument term is undefined, the application of a predicate does not hold, and the application of an operation is undefined (as usual in partial algebra). Definedness of terms also affects the holding of other atomic formulae: an existential equation holds when both terms are defined and equal, whereas a strong equation holds when they are both defined and equal, or both undefined.

Sort Generation Constraints: $(S', F') \in \mathbf{Sen}(\Sigma)$, with $(S', F') \subseteq (S, F)$, where $F = TF \cup PF$: A sort generation constraint is a further kind of Σ-sentence (in general not expressible as a first-order sentence). It is satisfied in a model when the carriers of sorts in S' are generated by functions in F' (and possibly from the carriers of sorts in $S \setminus S'$).

Institution: CASL signatures come equipped with *signature morphisms*. A signature morphism $\sigma : \Sigma \to \Sigma'$ between signatures $\Sigma = (S, TF, PF, P, \leq)$ and $\Sigma' = (S', TF', PF', P', \leq')$ maps sorts S to sorts S' so that the subsorting preorder is preserved, operation symbols $TF \cup PF$ to operation symbols $TF' \cup PF'$ so that the profiles, overloading and totality of operation symbols are preserved, and predicate symbols P to predicate symbols P' so that their profiles and overloading are preserved. With the obvious composition and identities, this defines the *category* **Sign** *of* CASL *signatures.*

A signature morphism $\sigma : \Sigma \to \Sigma'$ determines a translation of sentences $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ defined as usual (by substituting symbols from Σ' for symbols from Σ as determined by σ), and a reduct functor $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$, given in the usual manner as well. These mappings are functorial, defining functors $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ and $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{Cat}$.[7] Translation along signature morphisms preserves satisfaction: given $\sigma : \Sigma \to \Sigma'$, $M' \in \mathbf{Mod}(\Sigma')$ and $\Phi \in \mathbf{Sen}(\Sigma)$, $M' \models \mathbf{Sen}(\sigma)(\Phi)$ iff $\mathbf{Mod}(\sigma)(M') \models \Phi$. Thus, the above definitions determine the CASL *institution* [30].

In fact, the *subsorted* CASL institution outlined above may be reduced to an ordinary many-sorted CASL institution without subsorting, by replacing subsort pre-orders by explicit embeddings. A CASL signature $\Sigma = (S, TF, PF, P, \leq)$ reduces to a many-sorted first-order theory with the signature $\widehat{\Sigma} = (S, TF \cup Emb, PF \cup Proj, P \cup Memb)$, where $Emb = \{emb_{s,s'} \mid s \leq s'\}$ is a set of total operation symbols for subsort embeddings, and the sets $Proj$ (of projections onto subsorts) and $Memb$ (of subsort membership predicates) are defined similarly, and a set of first-order axioms that express the required properties of subsort embeddings and their interrelations with projections, subsort memberships and overloaded operations. Then CASL Σ-models coincide with many-sorted models of the resulting theory, and CASL Σ-sentences may be directly replaced by the corresponding $\widehat{\Sigma}$-sentences. It can be easily verified that this defines a *simple map* [42] between the two CASL institutions considered (with and without subsorting, respectively), see [49].

Semantic Functions: In the CASL *institution*, applications of predicates and operations in atomic formulae and terms are fully qualified by their profiles, so there is no overloading at that level. In contrast, basic specifications in the CASL *language* allow these profiles to be omitted, since they are usually evident from the context. In general, there may be many ways—but possibly none at all—of expanding an atomic formula in CASL by inserting profiles to give a well-sorted fully-qualified atom for constructing a sentence of the underlying institution. The atomic formula is well-formed when its expansion is unique (up to the commuting of embeddings with overloaded operations); the axioms of a well-formed basic specification determine a set of sentences of the CASL institution.

The semantics of a well-formed basic specification in CASL is given by a sig-

[7] As usual, **Set** and **Cat** denote the categories of all sets and of all categories, respectively.

nature Σ together with the class of those models $M \in \mathbf{Mod}(\Sigma)$ that satisfy all the sentences determined by the specification.

3.3 Language Constructs

This section provides examples that illustrate the CASL language constructs for use in basic specifications: declarations and definitions (of sorts, operations, predicates, and datatypes), sort generation constraints, and axioms (involving variable declarations, quantifiers, connectives, atomic formulae, and terms). The examples are shown in *display format*; for input, a suggestive (ASCII or ISO Latin-1) plain text approximation is used, e.g., '$\rightarrow$' is input as '`->`', and '$\forall$' is input as '`forall`'. Note that CASL allows declarations to be interspersed with definitions and axioms. Visibility is linear: symbols have to be declared before they can be used (except within datatype declarations, where non-linear visibility allows mutually-recursive datatypes—e.g., *List* and *NeList* in Sect. 3.1 above).

Sorts: Several sorts may be declared at once, possibly as subsorts of some other sort (written '<'):

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

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

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

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

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

ops $0 : Nat$;
$suc : Nat \rightarrow Pos$;
$\_\_ - \_\_ : Nat \times Nat \rightarrow ?\ Nat$;
$\_\_ + \_\_ : Nat \times Nat \rightarrow Nat$, **assoc**, **comm**, **unit** 0

The declaration of a partial function

op $pre : Nat \rightarrow ?\ Nat$

allows terms such as $pre(pre(suc(x)))$ to be well-formed. Whether or not this term denotes a value depends on the value of x (and on the model considered); it may sometimes be possible to infer from the axioms and declarations in a given specification that the value of a term involving partial functions is defined in all the models of the specification.

On the other hand, the declaration of *pre* as a total function

op $\quad pre : Pos \rightarrow Nat$;

enables subsort analysis and automatic propagation of definedness: a term such as $pre(suc(0))$ is well-formed whereas $pre(0)$ and $pre(pre(suc(0)))$ are ill-formed. The term $pre(pre(suc(suc(0))))$ is ill-formed as well, but inserting an appropriate cast (written using the reserved word 'as') makes it well-formed: $pre(pre(suc(suc(0)))\ \mathit{as}\ Pos)$. Note that both declarations of *pre* may coexist; the composition of the subsort embedding from *Pos* to *Nat* with the partial *pre* operation is then required to be the same function as the total *pre* operation, so that the value of $pre(suc(x))$ is independent of overloading resolution [14,51]. Operations may also be written with explicit qualification, e.g., $pre(n)$ may be written as $(op\ pre : Nat \rightarrow?\ Nat)(n)$. Sorted terms (interpreted as explicit applications of the appropriate subsort embeddings) are written straightforwardly, e.g. $suc(suc(n) : Nat)$. However, $suc(suc(n))$ is well-formed here as well, as the required subsort embedding can unambiguously be added by the subsort analysis.

So-called *mixfix notation* is allowed: place-holders for arguments are written as *pairs* of underscores (single underscores are treated as letters in identifiers). All symbols should be input in the ISO Latin-1 character set, but *annotations* [8] may cause them to be displayed differently, e.g., as mathematical symbols. In simple cases, operations may also be defined at the same time as they are declared:

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

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

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

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

[8] An annotation is an auxiliary part of a specification, for use by tools, and not affecting the semantics of the specification.

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

Datatypes: A datatype declaration looks like a context-free grammar in a variant of BNF. It declares the symbols on the left of '::=' as sorts, and for each alternative on the right it declares a constructor—possibly with some selectors. Such a declaration does not introduce any constraints other than the expected relationship between constructors and selectors.

type $\mathit{Collection} ::= \mathit{empty} \mid \mathit{just}(\mathit{Elem}) \mid \mathit{join}(\mathit{Collection}; \mathit{Collection})$

However, when datatypes are declared as 'free', the sorts declared freely extend those introduced earlier: distinct constructor terms of the same sort yield distinct values, and each declared sort is generated by its constructors. In the example of *Pair* below, *left* and *right* are declared as selectors yielding the respective arguments of the constructor *pair*.

free type $\mathit{Bit} ::= \mathit{0} \mid \mathit{1}$
free type $\mathit{Pair} ::= \mathit{pair}(\mathit{left}, \mathit{right} : \mathit{Elem})$

When there is more than one alternative in a datatype declaration, selectors are usually *partial* and then they should be declared as such, by inserting '?': [9]

free type $\mathit{Nat} ::= \mathit{0} \mid \mathit{suc}(\mathit{pre} :? \mathit{Nat})$

Subsorts in Datatype Declarations: The explicit introduction of subsorts in datatype declarations avoids partial selectors, as in the following alternative declaration of *Nat*, where *pre* is a total function from *Pos* to *Nat*, cf. the discussion above:

free types $\mathit{Nat} ::= \mathit{0} \mid$ **sort** $\mathit{Pos};$

$\mathit{Pos} ::= \mathit{suc}(\mathit{pre} : \mathit{Nat})$

The example also illustrates non-linear visibility within a list of datatype declarations: *Pos* is used before it is declared. Using subsorts such as *Pos*, other functions can now also be declared as total, such as:

op $\_\_\mathit{div}\_\_ : \mathit{Nat} \times \mathit{Pos} \rightarrow \mathit{Nat}$

[9] Constructors can also be declared to be partial, by inserting '?' after the list of argument sorts.

Overloading of functions (and predicates) can be used to extend existing definitions, as in:

$$\begin{array}{lll}
\textbf{free types} & \mathit{Int} ::= \textbf{sort } \mathit{Nat} \mid \textbf{sort } \mathit{Neg}; & \\
 & \mathit{Neg} ::= -\_\_(\mathit{Pos}) & \\
\textbf{ops} & \_\_+\_\_ : \mathit{Int} \times \mathit{Int} \rightarrow \mathit{Int}, \textbf{assoc}, \textbf{comm}, \textbf{unit } \mathit{0}; & \\
 & \_\_\mathit{div}\_\_ : \mathit{Int} \times \mathit{Pos} \rightarrow \mathit{Int}; & \\
 & \_\_\mathit{div}\_\_ : \mathit{Int} \times \mathit{Neg} \rightarrow \mathit{Int} &
\end{array}$$

The subsort Neg is freely constructed by the unary (prefix) constructor operation $-\_\_ : \mathit{Pos} \rightarrow Neg$ (the inverse operation): so, for each $n \in \mathit{Pos}$ we have a distinct $-n \in Neg$. Then, Int consists of Nat, i.e. elements of the subsort Pos and the constant 0, and the subsort Neg. For div, the use of proper subsorts excludes 0 and thus avoids partiality; the proper profile is chosen or, in case of 0, erroneous application is flagged by static analysis.

Sort Generation Constraints: The CASL syntax also allows datatypes to be declared as generated, so that the sorts are constrained to be generated by their constructors (and subsort embeddings):

$$\begin{array}{ll}
\textbf{generated type} & \mathit{Collection} ::= \mathit{empty} \mid \mathit{add}(\mathit{Elem}; \mathit{Collection}) \\
\textbf{forall} & x, y : \mathit{Elem}; c : \mathit{Collection} \;\bullet\; \mathit{add}(x, (\mathit{add}(y, c)) = \mathit{add}(y, (\mathit{add}(x, c))
\end{array}$$

In the case of a generated type (in contrast to a free type), axioms such as that above may still be added (thus forcing 'confusion' between constructor terms, which is not excluded by generation constraints); in both cases there are no values beyond those generated (no 'junk').

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

$$\begin{array}{l}
\textbf{generated} \\
\{\ \textbf{sorts } \mathit{Pos} < \mathit{Nat}; \\
\quad \textbf{ops } \mathit{0} : \mathit{Nat}; \quad \mathit{suc} : \mathit{Nat} \rightarrow \mathit{Pos}\ \}
\end{array}$$

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

$$\begin{array}{ll}
\textbf{vars} & m, n : \mathit{Nat}; p : \mathit{Pos} \\
\textbf{axioms} & \mathit{0} \le n; \neg(p \le \mathit{0}); \mathit{suc}(m) \le \mathit{suc}(n) \;\Leftrightarrow\; m \le n; \ldots
\end{array}$$

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

$\textbf{forall}\ x, y, z : \mathit{Elem}$

- $x \leq x$ %(reflexivity)%
- $x \leq y \wedge y \leq z \Rightarrow x \leq z$ %(transitivity)%

or within a formula using explicit quantification:

$\forall n : \mathit{Nat} \bullet \exists m : \mathit{Nat} \bullet n < m$
$\forall p : \mathit{Pos} \bullet \exists! n : \mathit{Nat} \bullet \mathit{suc}(n) = p$ %% exists uniquely

Casl allows to annotate axioms by labels. Take for instance the label '%(reflexivity)%' in the above example. It is also possible to write comments, e.g. '%% exists uniquely'.

The logical connectives have their usual interpretation:

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

In addition to the use of *if* as syntactic sugar for reverse implication, there is a conditional construct for terms:

$\mathit{abs}(x) = -x\ \mathit{when}\ x < 0\ \mathit{else}\ x$

Atomic Formulae: Definedness assertions can be explicit using *def* as in

$\mathit{def}\ \mathit{pre}(\mathit{suc}(n)) \wedge \neg \mathit{def}\ \mathit{pre}(0)$

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

$\mathit{def}\ \mathit{pre}(n) \Rightarrow \mathit{suc}(\mathit{pre}(n)) \overset{e}{=} \mathit{pre}(\mathit{suc}(n))$

Strong equations can be used to define partial functions inductively:

op $\_\_!\_\_ : \mathit{List} \times \mathit{Nat} \rightarrow ?\ \mathit{Elem};$
forall $n : \mathit{Nat}; L : \mathit{List}; x : \mathit{Elem}$
- $\neg \mathit{def}\ \mathit{nil}!n$
- $\mathit{cons}(x, L)!\mathit{0} = x$
- $\mathit{cons}(x, L)!\mathit{succ}(n) = L!n$

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

$n \in \mathit{Pos} \Leftrightarrow \mathit{def}\ \mathit{pre}(n)$

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

Further examples of basic specification constructs may be found in the appendices of the CASL Summary [21] and in [64,52,62]; see also Section 7.

4 Structured Specifications

The structuring features of CASL do not depend on the details of the features for basic specifications, so this part of the design is orthogonal to the rest. An important consequence of this is that sub-languages and extensions of CASL can be defined by restricting or extending the language of basic specifications (under certain conditions) without the need to reconsider or change the rest of the language.

CASL provides ways of building complex specifications out of simpler ones (the simplest ones being basic specifications) by means of various *specification-building operations*. These include translation, hiding, union, and both free and loose forms of extension. A structured specification denotes a class of CASL models over a signature determined by the specification, as with basic specifications. Thus the structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style.

Structured specifications may be named and a named specification may be *generic*, meaning that it declares some *parameters* that need to be *instantiated* when the specification is (re)used. Instantiation is a matter of providing an appropriate *argument specification* together with a *fitting morphism* from the parameter to the argument specification. Fitting may also be accomplished by the use of named *views* between specifications. Generic specifications correspond to what is known in other specification languages as (*pushout-style*) *parametrized specifications* [71].

4.1 Pragmatic Issues

Imposing No Structure on Models: The task of structuring requirement specifications at the early stages of development is quite different from specifying the architecture of an implementation: typically, rather small fragments or individual properties are put together. The crucial point is that structuring a specification at this stage does *not* impose any specific way of building its models. The models of structured specifications are of exactly the same kind as for basic specifications, just interpreting the symbols declared so that all the asserted properties are satisfied. Consequently, they do not in any direct way reflect the structure of the specification.

For example, consider a specification of the integers. One might choose to structure it as an extension of a specification of natural numbers (as in an example above), or to give it as a single basic specification. This choice does not affect the semantics of the specification, which in either case determines the same class of CASL models over the same signature: neither the signature nor the models reflect the structure of the extension.

Section 5 explains the 'architectural' specifications of CASL, which do allow one to prescribe the way the models are to be built from other models, thus specifying the 'structure-in-the-large' of models.

Names of Symbols: A general principle underlying the CASL design is '*same name, same thing*'. Thus when one sees two occurrences of the same sort in the same basic specification, one may be sure that they are always interpreted as the same carrier set. For operations and predicates, the situation is a little more subtle: the 'name' of an operation (or predicate) includes its profile of argument and result sorts, so there need be no relationship at all between say, '+' on integers and '+' on lists or sets, at least in the absence of subsorting.

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

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

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

Generic Specifications: A specification definition *names* a specification, allowing reuse by reference to the name. For example, Int might refer to a specification of the integers. In Casl, a named specification may also have parameters, intended to vary between references; the specification body is an extension of what is specified in the parameters, and the named specification is called *generic*. Each reference to a generic specification requires instantiation of all its parameters. For example (cf. the paragraph on Generic Specifications and Parameters in Section 4.3 below), List might refer to a specification that extends a parameter specification named Elem; any reference to List has to provide an argument specification that 'fits' Elem.

Note that generic specifications in Casl are *not* intended for defining arbitrary functions on specifications, unlike in some other frameworks such as ASL [70]—the Casl user is expected to express the structure of specifications directly using the Casl language constructs that are provided for that purpose.

4.2 Semantic Concepts

Institution Independence: The semantics of structured specifications inherits the notions of signature and class of models from the Casl institution as presented for basic specifications. However, the structuring part of Casl is independent of the details of basic specification: the same structuring may be used regardless of whether basic specifications are restricted in a sublanguage (e.g., by eliminating partial functions, subsorts, predicates, or explicit quantifiers) or extended (e.g., to allow higher-order functions). The semantics of Casl specification-building operations may essentially be given in an arbitrary institution, so the specification structuring mechanisms of Casl make sense as long as the semantics of basic specifications is based on an institution. This is much as in earlier approaches to specifications in an arbitrary institution, cf. [67], where the semantics of specification-building operations is given in terms of the constructions available in an arbitrary institution (with a prominent role played by reduct functors induced by signature morphisms, and by the categorical structure of model classes).

The only exception in Casl to strict institution-independence is the way names and their maps (forming signature morphisms) are handled. The standard notion of an institution of [30], and consequently the specification-building operations described e.g. in [67], take signature morphisms for granted. With the emphasis in Casl on the use of names of symbols (the 'same name, same thing' principle discussed above) this is not sufficient. Therefore, we work with *institutions with symbols* [47], which are institutions additionally equipped with a proper concept of symbol name: essentially, the category of signatures is provided with a functor to **Set** that assigns to each signature the *set of its symbols* and turns signature morphisms into symbol maps. This is used in Casl to analyze symbol lists and symbol maps, and to build signature morphisms out of them. Once this is done, the standard institution-independent semantic constructs are employed. Consequently, the semantics of Casl structuring mechanisms is independent from the particular institution with symbols in use.

Local Environments and Closed Specifications: In a specification, the so-called *local environment* records the symbol declarations that are currently visible. For basic specifications, visibility is linear (except within lists of datatype declarations) so the local environment merely grows as one proceeds through the declarations. For structured specifications, however, the local environments at different places may be completely unrelated. In fact, Casl structured specifications are always 'specification extensions' in principle, built over some local environment modelled as a signature that provides declarations external to the specification. Well-formed specifications that are built over the empty environment, which are therefore *self-contained* in the sense that they contain declarations of all the components (sorts, operation and predicate symbols) that they use, are called *closed*.

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

4.3 Language Constructs

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

Translation and Reduction: Translation of declared symbols to new symbols is specified straightforwardly by giving a list of 'maplets' of the form

$old \mapsto new$.

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

Identity maplets $old \mapsto old$ may be abbreviated to *old*, or simply omitted altogether. Optionally, the nature of the symbols concerned (sorts, operations, predicates) may be indicated by inserting the corresponding keywords.

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

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

> NAT **hide** Pos, suc

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

Unions and Extensions: The signature of a *union* of two (or more) specifications is the union of their signatures (defined componentwise, except that the union of the subsorting pre-orders must be further transitively closed). Given models over component signatures, the unique model over the union signature that extends each of these models is called their *amalgamation*; a tuple of models is called *compatible* if their amalgamation exists. Clearly, not all tuples of models over component signatures amalgamate: an obvious necessary condition is that the models coincide on the common symbols (including subsort embeddings) of the component signatures. However, even then it may be impossible to build their amalgamation; the trouble is that newly emerged (by transitive closure) subsort embeddings need not in general be compatible with each other and with the overloaded operations in the sense sketched in Sect. 3.2. Devising 'static' conditions that are as weak as possible but ensure compatibility of models is a topic of current research; the CASL semantics provides some such conditions—see [24] for details.

The models of a union are all amalgamations of the models of the component specifications. There are two extremes: when the specifications have disjoint signatures, the models of their union are essentially all tuples of the models of the component specifications; when they have the same signature, the union provides the *intersection* of the model classes, giving all models that satisfy both the specifications at once. For example, the signatures of NAT and STRING might be disjoint, so models of

NAT **and** STRING

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

MONOID **and** COMMUTATIVE

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

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

NAT **then**
  sort *Nat* < *Int*;
  ops $\_\_ + \_\_ : \mathit{Int} \times \mathit{Int} \rightarrow \mathit{Int}$;
    ...

or merely require further properties of old ones:

COLLECTION **then**
  forall $c : \mathit{Collection} \bullet \mathit{join}(c, c) = c$

Extensions can be classified by their effect on the model class specified. For instance, an extension is called *conservative* when no models are lost: every model of the specification being extended is a reduct of some model of the extended specification. CASL provides annotations %**implies** , %**def** , and %**cons** to denote that the model class is not changed, that each model of the specification can be uniquely extended to a model of the extended specification, or that the extension is conservative, respectively. It is important to note that these annotations have no effect on the semantics of a specification: a specifier may use them to express his intentions, tools may use them to generate proof obligations. Discharging these proof obligations increases the trustworthiness of a specification.

Free Specifications: The simplest case of a free specification is when the specification constrained to be interpreted freely is closed. The signature of the specification is unchanged, but the models are restricted to (the isomorphism class of) its initial models. For instance, the only models of the following specification are the standard models of Peano's axioms:

free
{ sort *Nat*; **ops** $0 : \mathit{Nat}$; $\mathit{suc} : \mathit{Nat} \rightarrow \mathit{Nat}$ **}**

The conciseness and perspicuity of such specifications may account for the popularity of frameworks that support initiality. When axioms are restricted to positive conditional existential equations, initial models of basic specifications always exist. More generally, a free specification may be a free extension, e.g.:

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

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

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

```
spec PartialOrder =
  sort Elem
  pred __ ≤ __ : Elem × Elem
  forall x, y, z : Elem
  •  x ≤ x                      %(reflexivity)%
  •  x = y if x ≤ y ∧ y ≤ x     %(antisymmetry)%
  •  x ≤ z if x ≤ y ∧ y ≤ z     %(transitivity)%

spec TotalOrder =
  PartialOrder
then
  forall x, y : Elem
  •  x ≤ y ∨ y ≤ z              %(comparability)%
```

Generic Specifications and Parameters: A parameter is a closed sub-specification—typically a reference to a rather simple named specification such as Elem. A generic specification is an extension of all its parameters.

```
spec Elem = sort Elem

spec List [Elem] =
  free type List ::= nil | cons(Elem; List)
```

A reference to a generic specification is called an *instantiation*, and has to provide an argument specification for each parameter, indicating how it 'fits' by giving a map from the parameter signature to the argument signature, e.g.:

LIST [NAT **fit** $Elem \mapsto Nat$]

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

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

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

Note that the 'same name, same thing' principle is maintained here. Moreover, to use the same sort name (say *Elem*) in both parameters would require some way of disambiguating the different uses of the name in the body, similar to an explicit renaming. Sharing of symbols between the body of a generic specification and its arguments in an instantiation is restricted to explicit *imports*, indicated as '**given**':

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

Had NAT been merely referenced in the body of LISTLENGTH, an instantiation such as LISTLENGTH [NAT] would be ill-formed. Well-formed instantiations always have a 'push-out' semantics. The models of such instantiations are amalgamations of models of the parameters and of the generic specification translated by the appropriate extension of the fitting morphism (see [24] for details).

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

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

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

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

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

Local Specifications: CASL also facilitates the hiding of auxiliary symbols by allowing the local scope of their declarations to be indicated. For instance, *insert* below is an auxiliary symbol for use in specifying *order*. The example illustrates a complete structured specification definition. *Elem* and the predicate $\_\_ \leq \_\_$ are declared in TOTALORDER above; $\_\_ \leq \_\_$ is used in the compound identifier $order[\_\_ \leq \_\_]$ to allow instantiations with a particular order such as $order[lexicographicOrder]$ later on. To show an alternative notation, the list constructor is declared as an infix operator $\_\_::\_\_$.

```
spec LISTWITHORDER [TOTALORDER] =
   free type $List[Elem] ::= nil \mid \_\_ :: \_\_(Elem; List[Elem])$
then
   local
       op $insert : Elem \times List[Elem] \to List[Elem]$;
       forall $x, y : Elem; l : List[Elem]$
     •  $insert(x, nil) = x :: nil$;
     •  $insert(x, y :: l) = x :: insert(y, l)$ when $x \leq y$
                        else $y :: insert(x, l)$
   within
       op $order[\_\_ \leq \_\_] : List[Elem] \to List[Elem]$
       forall $x : Elem; l : List[Elem]$
     •  $order[\_\_ \leq \_\_](nil) = nil$;
     •  $order[\_\_ \leq \_\_](x :: l) = insert(x, order[\_\_ \leq \_\_](l))$
end
```

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

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

view TO_IN_NAT : TOTALORDER **to** NAT =
  sort *Elem* $\mapsto$ *Nat*, **pred** $\_\_ \leq \_\_ \mapsto \_\_ \leq \_\_$

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

LISTWITHORDER [**view** TO_IN_NAT]

makes it clear that the argument is not merely a named specification with an implicit fitting map, which would be written simply LISTWITHORDER [NAT]. The rules regarding omission of 'evident' maps in explicit fittings apply to named specification morphisms too.

A more extended example may be found in section 7.

5 Architectural Specifications

Architectural specifications in CASL are for describing the modular structure of software, in contrast to structured specifications where the structure is only for specification presentation purposes. Architectural specifications are probably the most novel aspect of CASL; they are not entirely new, but they have no counterpart in most algebraic specification languages. An architectural specification consists of a list of *unit declarations*, indicating the component modules required with specifications for each of them, together with a *unit expression* that describes the way in which these modules are to be combined. [10]

As the above terminology indicates, we have chosen to avoid the overloaded term 'module' and its direct connotations with various constructs of programming languages, using in CASL the term 'unit' instead. *Units* in CASL may be either *simple* (self-contained, non-generic) and then semantically they are simply CASL models; or they may be *functional* (generic). Functional units are functions which map (tuples of compatible) CASL models to CASL models. These functions are required to be *persistent*, meaning that the result model expands the argument model, which corresponds to the fact that a software module must use its imports as supplied without altering them.

Of course, the idea is that eventually in the process of systematic development of modular software from specifications, units are implemented as software

[10] There is an unfortunate potential for confusion here: in CASL, the term 'architecture' refers to the 'implementation' modular structure of the system rather than to the 'interaction' relationships between modules in the sense of [1].

modules (or pieces of code encapsulated in one way or another) in some chosen programming language. However, this step is beyond the scope of specification formalisms, so in CASL and in this paper we identify units with models and model functions, as indicated above. The modular structure of the software system under development, as described by an architectural specification, is therefore captured here simply as an explicit, structural way to build CASL models.

5.1 Pragmatic Issues

Reusability: Whereas *structuring of specifications* into unions, extensions, instantiations of generic specifications, etc., encourages the reuse of parts of specifications, it does *not* affect the models at all, and the monolithic result of implementing a structured specification—its specific model—is unlikely to be reusable. Architectural specifications allow the components of such an implementation to be described separately, supporting reusability at the software component level.

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

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

The corresponding architectural specification in CASL requires one to provide the units N and F, and builds their composition. If the model N of NAT is subsequently changed, F may be reused, and does not have to be rebuilt. (F may also be changed without changing N, of course.)

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

Decomposition/Composition: A crucial aspect of architectural specifications is that they provide decompositions of possibly large and complex development tasks into smaller sub-tasks—as well as indicating how to compose, or link together, the results of sub-tasks. A unit specification expresses everything that those who are implementing it (building a model) and those who are using it (to build further models) need to know.

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

5.2 Semantic Concepts

Architectural Models: An architectural specification denotes a class of *architectural models* which consist of:

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

As mentioned above, units are either CASL models, or functions from CASL models to CASL models (higher-order units are envisaged but currently not included in CASL). The unit functions are always persistent, so that the results extend the unmodified arguments (the function F considered above should clearly not be allowed to ignore the argument implementation N and incorporate a different implementation of NAT). When unit functions have more than one argument, the arguments must be compatible, in particular implementing any common symbols in exactly the same way—this follows immediately from the requirement that a function should extend each argument separately.

Unit Specifications: A specification of a simple unit is any structured specification, with its usual semantics as a class of models; a unit *satisfies* such a specification if it belongs to this class. Specifications of functional units provide a specification of an argument and its extension to a specification of the result. Its denotation is the class of all persistent functions that map models of the argument specification to models of the result specification. This extends naturally to multi-argument functional units: their specification involves a tuple of specifications for the arguments, and the persistent functions in their denotations take compatible tuples of models of these specifications as arguments. As before, a functional unit (which is a function on models) *satisfies* such a specification if it belongs to the class of functions the specification denotes.

The above semantics of specifications of functional units should be contrasted with the semantics of generic specifications. In CASL, generic specifications are just (named, closed) structured specifications with an indicated parameter part. However, via the semantics of their instantiation, they may be viewed as functions from (argument) specifications to (result) specifications, which semantically amounts to functions that map classes of models to classes of models. Of course, as discussed in [65], there are close (Galois) connections between functions on classes of models and classes of functions on models, but still, these are quite different mathematical objects. Generic specifications and specifications of functional units are quite different concepts, occurring at different levels of CASL, with their different roles in software specification and development.

Institution Independence: As with structured specifications (cf. Sect. 4.2), the design of CASL architectural specifications is largely independent from the underlying CASL institution with symbols. However, some details of their semantics, notably concerning conditions to ensure compatibility of models and the issues of sharing components between models (see [11]), require additional information about the signatures and models considered. An appropriate precise notion of *institution with symbols and sharing* is currently under development to provide a basis for a completely institution-independent semantics of CASL architectural specifications.

5.3 Language Constructs

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

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

```
**arch spec** IMPNATLIST =
  **units** $N$ : NAT ;
        $F$ : NAT $\rightarrow$ LIST [NAT]
  **result** $F[N]$
```

An architectural model of the above architectural specification consists of:

- a unit N that is a model of Nat;
- a unit function F that satisfies Nat $\rightarrow$ List [Nat], which is a functional unit specification where Nat specifies arguments and List [Nat] specifies results of the functional units; and
- the unit $F[N]$, which is a model of the structured specification List [Nat].

Unit Declarations and Definitions: A unit declaration names a unit that is to be developed, and gives its specification, which may be either a structured specification for simple units, or a specification of functional units, as discussed above. Some examples of unit declarations:

> N : Nat
> F : Nat $\rightarrow$ List [Nat]
> L : List [Nat] **given** N

The form of unit declaration using '**given**' provides an implicit declaration of a functional unit that gets applied just once (in this case, to N). If the declaration of F in the architectural specification ImpNatList given above were to be replaced by that of L above (letting the result be simply L as well) then architectural models of the resulting architectural specification would still provide a functional unit that gives a model of List [Nat] extending any model N of Nat.

A unit definition names a unit that can be constructed from previously introduced units (in the same architectural specification) as determined by a unit expression:

> $L = F[N]$
> $L' = F[N$ **fit**...] **hide**...

The unit expression on the right-hand side of a unit definition is of the same form as the result unit of an architectural specification (see below).

Unit Specifications: Unit specifications can be named, allowing them to be reused. For instance:

> **unit spec** GenList = Nat $\rightarrow$ List [Nat]

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

Architectural specifications themselves may also be used as unit specifications, describing the class of units that are the result units in their architectural models.

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

- amalgamation of compatible simple units: N **and** C
- application of functional units to compatible arguments, via a fitting morphism if necessary: $F[N]$, $F[N$ **fit**$\ldots]$
- abstraction: $\lambda N : \textsc{Nat} \bullet \ldots N \ldots$
- reduction of simple units: U **hide** $\ldots$, U **reveal** $\ldots$
- translation of simple units: U **with** $\ldots$

However, the semantics of unit expressions involves operations on individual models, rather than on entire model classes. In particular, amalgamation of models requires their compatibility (a sufficient static condition to ensure compatibility, hinted at in Sect. 4.3, is checked). Amalgamation and hence checking compatibility are also involved in the semantics of application, where the result amalgamates the argument model with the appropriately translated result of the direct application of the functional unit to the argument reduced by the fitting morphism. Thus application here conforms with pushout-style instantiation of generic specifications. Abstraction builds a functional unit using λ-notation with the usual meaning; this is needed to allow architectural specifications whose results are unit functions. Reduction of simple units are direct applications of the model reduct functor determined by the signature morphism extracted from the given symbol lists and mappings. Translation is an inverse construction to the reduct, somewhat complicated in case of non-injective renaming by necessary additional requirements to make it well-defined, similar to the compatibility of models necessary for amalgamation.

5.4 Example

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

```
arch spec CN_LIST =
   units  C : CHAR ;
          N : NAT ;
          F : ELEM → LIST[ELEM]
   result F[C fit Elem ↦ Char] and F[N fit Elem ↦ Nat]
```

Further examples of architectural specifications are given in the CASL Summary [21] and in [11].

6 Libraries of Specifications

Libraries in CASL are collections of named closed basic and structured specifications, their views, architectural specifications, as well as unit specifications. A specification can refer to an item in a library by giving its name and the location of the library that contains it. CASL includes direct support for establishing distributed libraries on the Internet .

6.1 Pragmatic issues

When specifications are collected into libraries, the question of visibility of symbols between specifications arises. In CASL, the symbols available in a specification are only those that it declares itself, together with those declared (and not hidden) in named specifications that it explicitly references. Thus when a specification in a library is changed, it is straightforward to locate other specifications that might be affected by the changes.

Another issue concerns visibility of specification names. In CASL, visibility is linear: a specification may only refer to names of specifications and views that precede it in the library. The motivation for this restriction is partly methodological (the library is presented in a bottom-up fashion), partly from implementation considerations (a library can be processed sequentially), and partly from the difficulty of giving a satisfactory formal semantics to mutually-dependent specifications (some of CASL's specification-building operations are not monotone w.r.t. the inclusion of model classes, so the usual fix-point semantics would not work in general).

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

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

6.2 Semantic Concepts

The semantics of a library is a global environment that maps names of specifications, views, architectural and unit specifications (previously declared or downloaded in the same library) to their denotations.

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

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

6.3 Language Constructs

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

```
library OrderTheory
...
spec PartialOrder =
  ...
spec TotalOrder =
  PartialOrder
then
  ...
```

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

```
library Numbers
from OrderTheory get TotalOrder ↦ Order
spec Nat = ...
view Order_in_Nat : Order to Nat =
  sort Elem ↦ Nat, pred __ ≤ __ ↦ __ ≤ __
  ...
```

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

library NUMBERS **version** 1.2
from ORDERTHEORY **version** 1.0.1 **get** TOTALORDER
...

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

7 Extended Example

The following example shows specifications for the datatypes 'finite map' and 'array' in CASL. It has been taken from the library STRUCTUREDDATATYPES of the document on Basic Datatypes for CASL [64] and illustrates a list of structured specification definitions as they appear in a library. The library NUMBERS including the specifications NAT and INT is omitted here. The specification FINITESET is only indicated. The labels of axioms (which may be introduced as annotations for use with tools) are also omitted. Before presenting the specifications, some aspects from the underlying methodology (cf. [63]) are discussed.

Sort Generation: The specification of finite maps from sort S to sort T is divided into two parts: GENERATEFINITEMAP is concerned only with sort generation, while FINITEMAP deals with all additional aspects. As generation of sorts is a rather subtle part of a specification, this style hopefully avoids reader confusion.

Annotation %implies: The specification of arrays illustrates how to separate the definition of predicates and operators from the specification of their desired properties: the annotation **%implies** in the specification ARRAY indicates that the properties specified after the keyword **then** should follow—from the specifier's point of view—from the previous axioms. For example, modelling an array by finite maps yields the usual array axioms. Writing an annotation **%implies** leads to the generation of proof obligations. Discharging these obligations (with a verification tool) increases trust in the correctness of the specification.

library BASIC/STRUCTUREDDATATYPES **version** 0.4.1
%% authors: M.Roggenbach, T.Mossakowski, L.Schröder, 5.5.00

```
from Basic/Numbers version 0.4.1 get Nat, Int

spec FiniteSet [sort Elem] = . . . end

spec GenerateFiniteMap [sort S] [sort T] =
free {
  type FiniteMap[S, T] ::= [] | __[__/__](FiniteMap[S, T]; T; S)
  forall M : FiniteMap[S, T]; s, s1, s2 : S; t1, t2 : T
  •   M[t1/s][t2/s] = M[t2/s]
  •   M[t1/s1][t2/s2] = M[t2/s2][t1/s1] if ¬ s1 = s2 }
end

spec FiniteMap [sort S][sort T] given Nat =
  GenerateFiniteMap [sort S][sort T]
  and FiniteSet [sort S] and FiniteSet [sort T]
then
  free type Entry[S, T] ::= [__/__](target : T; source : S)
  preds
      isEmpty : FiniteMap[S, T];
      __ϵ__ : Entry[S, T] × FiniteMap[S, T];
      __ :: __− > __ : FiniteMap[S, T] × FinSet[S] × FinSet[T]
  ops
      __ + __, __ − __ : FiniteMap[S, T] × Entry[S, T] → FiniteMap[S, T];
      dom : FiniteMap[S, T] → FinSet[S];
      range : FiniteMap[S, T] → FinSet[T];
      __!__ : FiniteMap[S, T] × S →? T;
      __ ∪ __ : FiniteMap[S, T] × FiniteMap[S, T] →? FiniteMap[S, T]
  forall M, N, O : FiniteMap[S, T]; s, s1 : S; t, t1 : T; e : Entry[S, T];
      X : FinSet[S]; Y : FinSet[T]
  •   isEmpty(M) ⇔ M = []

  •   ¬ [t/s] ϵ []
  •   [t/s] ϵ M[t1/s1] ⇔ ([t/s] = [t1/s1] ∨ [t/s]ϵM)

  •   M :: X− > Y ⇔ dom(M) = X ∧ range(M) ⊆ Y

  •   M + [t/s] = M[t/s]

  •   [] − [t/s] = []
  •   (M + [t/s]) − [t1/s1] = M − [t1/s1] when [t/s] = [t1/s1]
      else( M − [t1/s1]) + [t/s]

  •   sϵdom(M) ⇔ ∃t : T • [t/s]ϵM
  •   tϵrange(M) ⇔ ∃s : S • [t/s]ϵM
  •   ¬def []!s
  •   (M + [t1/s1])!s = t1 when s = s1 else M!s
  •   M ∪ N = O ⇔ (∀e : Entry[S, T] • eϵO ⇔ (eϵM ∨ eϵN))
then %implies
  forall s : S; M : FiniteMap[S, T]
```

- $def\ M!s \Leftrightarrow s\epsilon dom(M)$

end

spec ARRAY [**ops** $min, max : Int$ **axiom** $min \leq max$] [**sort** $Elem$]
  given INT
$=$
  sort $Index = \{i : Int \bullet min \leq i \wedge i \leq max\}$
then
  { FINITEMAP [**sort** Index][**sort** Elem]
    with sort $FiniteMap[Index, Elem] \mapsto Array[Elem]$, **op** $[] \mapsto empty$
  then
    op $\_\_!\_\_ := \_\_ : Array[Elem] \times Index \times Elem \to Array[Elem]$;
    forall $A : Array[Elem]; i : Index; e : Elem$
    - $A!i := e = A[e/i]$

  } **reveal sort** $Array[Elem]$, **ops** $empty, \_\_!\_\_, \_\_!\_\_ := \_\_$
then %**implies**
  forall $A : Array[Elem]; i, j : Index; e, f : Elem$
  - $\neg def\ empty!i$
  - $def\ (A!i := e)!i$
  - $(A!i := e)!j = e\ if\ i = j$
  - $(A!i := e)!j = A!j\ if\ \neg(i = j)$

end

The example demonstrates overloading in CASL. In the specification FINITEMAP, the symbol + has different meanings: unary and binary plus on natural numbers, and adding an entry to a finite map. For the operator for combining two finite maps, the symbol $\cup$ was used instead since it has a slightly different semantics; technically, + could have been used as well.

The example makes use of several structuring constructs of CASL: in ARRAY, symbols of FINITEMAP are *translated*, e.g.

sort $FiniteMap[Index, Elem] \mapsto Array[Elem]$, **op** $[] \mapsto empty$

or *removed,* e.g.

reveal sort $Array[Elem]$, **ops** $empty, \_\_!\_\_, \_\_!\_\_ := \_\_$

FINITEMAP is structured as the *union* of

- GENERATEFINITEMAP[**sort** S][**sort** T],
- FINITESET[**sort** S], and
- FINITESET[**sort** T],

which is *extended* by several predicates and operations in the next step.

GENERATEFINITEMAP[**sort** S][**sort** T] introduces finite maps as a *free spec-*

ification.

ARRAY is a *generic specification.* In an instantiation

$$\text{ARRAY } [\textbf{op } n : \mathit{Int} \textbf{ fit ops } \mathit{min} \mapsto \mathit{1}, \mathit{max} \mapsto n] \ [\text{NAT}]$$

the sort name $\mathit{Array}[\mathit{Elem}]$ is effectively instantiated to $\mathit{Array}[\mathit{Nat}]$. This instantiation is consistent if $1 \leq n$, i.e. the array bounds fullfill the axiom of the parameter [**ops** $\mathit{min}, \mathit{max} : \mathit{Int}$ **axiom** $\mathit{min} \leq \mathit{max}$].

8 Other Algebraic Specification Languages

In this section we briefly compare CASL with a few other representative algebraic specification languages. One of the design goals of CASL as a *common* language was to provide a migration path for users of other languages, hence we pay special attention to features of these other languages that are not available in CASL. Further comments are in [49].

8.1 ASL

ASL is a minimalist kernel specification language containing a small number of simple but powerful constructs. More convenient and user-friendly specification constructs could be defined in terms of those supplied, as was done in an early draft of the semantics of Extended ML [66]. Various versions of ASL have been used; here we will refer to the institution-independent version defined in [67] with the extensions for specifying parametrized programs in [65]. Like CASL, this is an institution-independent specification language; unlike CASL there is no standard language defined for writing basic specifications, so we restrict attention to structured specifications and architectural specifications.

With the exception of ASL's observational abstraction operation, the specification-building operations in ASL are similar in power to those in CASL but less convenient: for instance, union is restricted to combining specifications having the same signature. Instead of free specifications, ASL provides an operation called **minimal** which is not equivalent but can be used to achieve similar aims, and [67] shows how freeness requirements could easily be added. Pushout-style generic specifications are not available in ASL although the same effect can be obtained using union and translation. Instead, parametrized specifications are formed in ASL by lambda-abstraction, with instantiation being simply beta-reduction; this is a more powerful parametrization mechanism that was not adopted in CASL for simplicity. Observational abstraction, which closes the models of a specification under observational equivalence, is not available in CASL because it greatly complicates the problem of reasoning about specifications, see e.g. [8], and because it was felt that it was more appropriately embedded in the relationship of specification refinement than provided in CASL itself, see [10].

ASL's specifications of parametrized programs (so-called 'Π-specifications') correspond to specifications of functional units in CASL architectural specifications. In ASL, parametrized specifications and Π-specifications (and the parametrized programs themselves) may be combined in a rather unconstrained way with no restriction to first-order parametrization. Certain combinations that are not available in CASL seem useful (see [2] for examples).

8.2 Larch

Larch [33,34] is a family of specification languages. Each Larch specification has components written in two languages: one designed for a specific programming language, the *Larch interface language*, and another common to all programming languages, the *Larch shared language* LSL. The interface languages provide a way of making assertions about program states, exceptions, etc., with the specification features available depending on the features of the programming language under consideration. LSL specifications define auxiliary higher-level abstractions from the problem domain for reference by interface specifications.

This 'two-tiered' approach to making the connection between programs and specifications is the main distinguishing feature of Larch. CASL has been designed to be independent of the programming language used to realize specifications and so the connection with programming languages is radically different. To use a programming language with CASL, it is necessary to provide a semantics for the language which assigns to each program P its denotation $[\![P]\!]$ as a CASL model; then P satisfies SP whenever $[\![P]\!]$ is a model of SP. (This semantics may be rather indirect, and would in general involve a non-trivial abstraction step. It has not yet been attempted for any real programming language.) There is a further connection at the level of architectural specifications: for each operation used in unit expressions to combine component units (application of functional units to arguments, etc.), we need to give a corresponding operation or combination of operations for combining modules in the programming language.

LSL itself is (by design) a much simpler specification language than CASL. Apart from equational axioms, there are sort generation constraints as in CASL and a way of asserting that a given list of operations constitutes a complete set of observers for a given sort. This is not available as a separate construct in CASL but it is easily expressible as an axiom. It is possible to claim that a given assertion (or set of assertions, including sort generation constraints etc.) is a consequence of a given specification, or that the definition of a given operation is sufficiently complete. The first of these can be expressed in CASL using the **%implies** annotation, while the second is an obvious candidate for another form of CASL annotation. The specification-building operations in LSL are limited to translation and union with no hiding construct or free specifications. The parametrization mechanism is merely a convenient syntax for renaming selected sorts and/or operations of a specification, with the appearance or non-appearance of parameters having no semantic significance.

8.3 ACT ONE and ACT TWO

ACT [17] is an approach to formal software development that includes a language called ACT ONE [16] for writing algebraic specifications with conditional equational axioms, and an extension called ACT TWO [27] for writing module specifications. ACT ONE has a pure initial algebra semantics in which specifications denote free functors. Its specification-building operations are similar to those in CASL, including pushout-style generic specifications with compound identifiers as in CASL, except that no operation for hiding is available. Instead, this is provided at the ACT TWO level where module specifications include import and export interfaces in the form of ACT ONE specifications extended by permitting first-order axioms. Module-building operations are module-level analogues to specification-building operations.

The most interesting point of comparison between CASL and ACT concerns the relationship between generic specifications and specifications of functional units in CASL architectural specifications. In CASL, these are very different: specification structure (as in generic specifications) is for presentation purposes only and imposes no structure on the models that are specified, while architectural specifications are precisely about the description of modular structure. This distinction, which originates in [65], is reflected in the semantics of CASL, as already explained in Sect. 5.2. Despite appearances, this is not the same as the distinction between ACT ONE and ACT TWO. But a similar distinction does arise in the semantics of ACT ONE, which is given at two levels, the specification level and the model level. The semantics of a generic specification at the specification level is the same as in CASL, while its semantics at the model level is a functor (the initial object in a class of functors), which is more closely related to the semantics of unit specifications in CASL.

8.4 OBJ3

OBJ3 [32] is an executable specification language, also known as an 'ultra-high-level' programming language. It is institution-independent in the sense that it originated as an implementation of Clear [12], which is institution-independent, for the institution of order-sorted conditional equational logic. OBJ3's mixfix notation for operations was the origin of mixfix notation in CASL and other languages. The same for views, although the features for views in OBJ3 differ from those in CASL. In an OBJ3 view, an operation may be mapped to a 'derived operation', for instance

$$x \leq y \mapsto x < y \vee x = y$$

On the other hand, views in CASL may have parameters to be instantiated when the view is used (see [21]) and these have been found to be useful in some circumstances; such parameters are not available in OBJ3 although they appear to be planned for a future release. The specification-building operations in OBJ3 are similar to those in CASL, except that no operation for hiding is provided.

The most interesting point of comparison between OBJ3 and CASL concerns the treatment of subsorts. Subsorts in algebraic specification originated in OBJ3, cf. [31], and the approach in OBJ3 influenced the design of CASL, but the approach taken in CASL is deliberately different, as already indicated in Sect. 3.1. First, a relatively minor point is that CASL interprets subsorts using embeddings between carriers rather than inclusions as in OBJ3, with inclusions being a special case. The difference between these is not significant in most examples, see Sect. 3.4 of [14]. OBJ3's approach to subsorts requires signatures to be 'regular' and 'coherent', while no such conditions are required in CASL which is convenient since they are not preserved by structuring operations in general. This requires a slightly more complicated treatment of overloading in CASL's definition of well-formed term than in OBJ3. Probably the most important discrepancy is that projection functions from supersorts to subsorts are regarded as partial functions in CASL. In OBJ3, they are total 'retract' functions which yield values that can be viewed as error messages when applied to values outside the subsort. The pros and cons of the two approaches are a matter of fierce debate, see [14] and [15] for the CASL point of view and [29] for the OBJ3 point of view. Some aspects of the semantics of subsorts in OBJ3 are unclear, see [49] for discussion. On the other hand, development of good tool support for reasoning about CASL specifications involving subsorts is still a research issue.

9 Foreground

The Common Framework Initiative has task groups on language design, semantics, tools, methodology, and reactive systems. There is a substantial amount of interaction between the task groups, which is supported by many of the CoFI participants being active in more than one task group. The overall coordination of these task groups was managed by Peter Mosses (Aarhus) from the start of CoFI in September 1995 until August 1998, and subsequently by Don Sannella (Edinburgh).

The European Commission has provided funding for the European component of CoFI as ESPRIT Working Group 29432 for two years starting October 1998, see `http://www.dcs.ed.ac.uk/home/dts/CoFI-WG/`. The partners are the coordinating sites of the various CoFI task groups (University of Bremen, Warsaw University, Ecole Normale Supérieure de Cachan, INRIA Lorraine, University of Genova, University of Aarhus) with the University of Edinburgh as overall coordinator. Its goals are: to coordinate the completion of and disseminate the Common Framework; to demonstrate its practical applicability in industrial contexts; and to establish the infrastructure needed for future European collaborative research in algebraic techniques. Before October 1998, CoFI relied on unfunded efforts by its participants, with initial support from the ESPRIT COMPASS Working Group 3264/6112 until that terminated in March 1996.

9.1 Language Design

The Language Design Task Group is coordinated by Bernd Krieg-Brückner, Bremen.

Until October 1998, the main language design task was finalization of the Casl design. The documentation of the final design is given by the Casl Language Summary [21]; a (now slightly outdated) rationale for the language design was published in 1997 [55]. The semantics, tools, and methodology task groups have all provided essential feedback regarding language design proposals.

Recent work on syntactic issues regarding mixfix parsing and syntactic extensions for literals to be used with Basic Datatypes for Casl [64,52,62] has now been completed.

Various interesting sublanguages of Casl, e.g., total, many-sorted, equational—mostly corresponding closely to embeddings of the specification languages of other frameworks into Casl [44–46,49]—have been defined. The logic underlying Casl has been translated to first-order logic (or second-order logic when sort generation constraints are considered). This allows the re-use of first-order and higher-order theorem provers for Casl [48,49]. Some extensions are now being investigated, in particular for higher-order [50] and object-oriented specifications. Possible extensions for specification of reactive systems are treated in a separate task group.

9.2 Semantics

The Semantics Task Group is coordinated by Andrzej Tarlecki, Warsaw.

The formal semantics of Casl, which is complete but whose presentation still requires some polishing, is given in [24]. The semantics is divided into the same parts as the language definition (basic specifications, structured specifications, etc.); each part is split further into *static semantics* and *model semantics.*

The *static semantics* checks well-formedness of phrases and produces a 'syntactic' object as result, failing to produce any result for ill-formed phrases. For example, for a basic specification the static semantics yields a *theory presentation* containing the sorts, function symbols, predicate symbols and axioms that belong to the specification. (Actually it yields an *enrichment*: when a basic specification is used to extend an existing specification it may refer to sorts, functions and predicates from the local environment.) A phrase may be ill-formed because it makes reference to non-existent identifiers or because it contains a sub-phrase that fails to type check.

The *model semantics* provides the corresponding model-theoretic part of the semantics, and is intended to be applied only to phrases that are well-formed according to the static semantics. A statically well-formed phrase may still be ill-formed according to the model semantics: for example, if a generic specification is instantiated with an argument specification that has an appropriate signature but which has models that fail to satisfy the axioms in the parameter specification, then the result is undefined. The judgements of the static and model semantics are defined inductively by means of rules in the style of Natural Semantics.

The orthogonality of basic specifications in Casl with respect to the rest of the language is reflected in the semantics by the use of a variant of the notion of institution [30] called an *institution with symbols* [47]. The semantics of basic specifications introduces a particular institution with symbols, and the rest of the semantics is based on an arbitrary institution with symbols.

The semantics provides a basis for the development of a proof system for Casl. As usual, at least three levels are needed: proving consequences of sets of axioms; proving consequences of structured specifications; and finally, proving the refinement relation between structured specifications. The semantics of Casl gives a reference point for checking the soundness of each of the proposed proof systems and for studying their completeness.

Apart from polishing the full semantics of Casl and from consideration of the semantics of sublanguages and extensions of Casl, the development of a proof system for Casl is the main work remaining for the semantics task group.

9.3 Methodology

The Methodology Task Group is coordinated by Michel Bidoit, Cachan.

The original motivation for work on algebraic specification was to enable the stepwise development of correct software systems from specifications with verified refinement steps. Casl provides good support for the production of specifications both of the problem to be solved and of components of the solution, but it does not incorporate a specific notion of refinement. Architectural specifications go some way towards relating different stages of development but they do not provide the full answer. Other methodological issues concern the 'endpoints' of the software development process: how the original specification is obtained in the first place (requirements engineering), and how the transition is made from Casl to a given programming language. Finally, the usual issues in programming methodology are relevant here, for instance: verification versus testing; software reuse and specification reuse; software reverse engineering; software evolution.

Casl has been designed to accommodate multiple methodologies. Various existing methodologies and styles of use of algebraic specifications have been considered during the design of Casl to avoid unnecessary difficulties for users who are accustomed to a certain way of doing things. For example, the methodology in [68] is being adapted to Casl.

The major task at the moment is the production of a user's guide for Casl. Moreover, various case studies are to be coordinated within this task group.

9.4 Tools

The Tools Task Group is coordinated by Hélène Kirchner, Nancy.

The aims of this task group are threefold:

- To provide a minimal but widely available set of tools for CASL, including syntax and static semantics checkers, library support, Emacs and LaTeX modes.
- To take advantage of and to reuse existing tools developed in the community for prototyping, testing, checking properties of programs, verifying the correctness of a specification or of a refinement step. Many of these are specialized, that is only applicable to a particular sub-language and its associated logics. Collaboration with the developers of tools for other languages will usually be needed to enable the use of CASL specifications with those tools.
- Ultimately to achieve a coherent and efficient integration of sub-languages and related tools. This raises the issue of combining and embedding different logics.

The CASL tool set (CATS, [48]) is an integrated set of tools combining a parser, a static checker, a LaTeX pretty printer and facilities for printing signatures of specifications and structure graphs of CASL specifications, with links to various verification and development systems. In addition, we plan to provide a structure editor, an Emacs mode, and a graphical interface to display the structure graphs. To experiment with CASL specifications, the CATS system provides different user interfaces: a Web-based interface, and a compact stand-alone version. A repository with successfully and unsuccessfully parsed specifications is under development.

CASL offers a flexible syntax including *mixfix* notation, which requires advanced parsing technology. ASF+SDF was used to prototype the CASL syntax in the course of its design, and several other parsers have been developed concurrently with the concrete syntax, which had the advantage of helping to detect ambiguities and inconsistencies in the syntax, cf. [73], [72], [51].

A LaTeX package for formatting CASL specifications has been developed [56]. This package is aimed at facilitating the pretty-printing and the uniform formatting of CASL specifications, and the easy combination of parts of documents written by different authors. An automatic conversion from LaTeX to HTML provides another widely available format for exchanging specifications through the Web.

Interoperability of CASL and existing tools is a major goal of the Tools group. The first step has been to propose an interchange (or interoperability) format that can be accepted as input and output by every tool. The starting idea was to adopt basically abstract syntax trees with annotations providing specific information to communicate with various tools (parsers, rewrite engines, proof tools, etc.). The Annotated Term (ATerm) Format described in [74] has been chosen as a common interchange format for CoFI tools. Work is in progress to also provide XML as an external interchange format. Based on either of these low-level formats, several high-level formats such as CasFix [75] (for abstract syntax trees of Casl specifications), CasEnv (for global environments containing signature information etc.) and FCasEnv (a flattened version of CasEnv, for use with tools that do not support structured specifications) have been developed. Formats for storing proofs and developments will follow.

Existing rewrite engines embedded in OBJ, ASF+SDF and ELAN provide a good basis for prototyping (parts of) Casl specifications. For instance, the ELAN compiler [43] efficiently supports many-sorted conditional rewrite rules with associative commutative functions. A first prototype has been realised that reads in the FCasEnv format, and translates it into EFix format (ATerms for ELAN) which can then be executed by the ELAN interpreter or compiled to produce C code [38].

The standalone version of CATS also contains an encoding into several other logics. The encoding transforms a Casl specification into second-order logic step by step. First, partiality is encoded via error elements living in a supersort; second, subsorting is encoded via injections; and third, sort generation constraints are expressed via second-order induction axioms. It is possible to stop after the first or second step if one wants to use a tool supporting subsorting or sort generation constraints directly. For details, see [49], where alternative encodings are also described. In this way, CATS allows to interface Casl with a large number of first- and higher-order theorem provers.

The HOL-CASL system, being built on top of CATS, uses the encoding of Casl into second-order logic to connect Casl to the Isabelle theorem prover and the generic graphical user interface IsaWin. This approach to encoding Casl in proof systems such as Isabelle or PVS allows verification and program transformation [51,49].

Various verification tools have already been developed for algebraic specifications, and can be reused for specific subsets of Casl: equational, conditional, full first-order logic with total functions, total functions with subsorts, partial functions, etc. The system INKA 5.0 [3] provides an integrated specification and theorem proving environment for a sub-language of Casl that excludes partial functions (with the encoding provided by CATS, it will also be useable with full Casl); a similar adaptation of the KIV [60] system is under way.

Currently, CATS is connected to the development graph management component of the INKA theorem proving system [3]. Structured Casl specifications in the CasEnv format are translated to development graphs [4]. The development graph supports the management of theories and proof obligations that arise from Casl specifications in a theorem prover-independent way. Moreover, it provides an efficient way of managing change, allowing re-use of those parts of proofs that are not affected by the change of a specification.

The next step is the integration of other existing tools, especially for prototyping and verification. Participants of the Tools group already have experience with tool integration, with Corba-IDL [37], the Tool Bus [7] developed in Amsterdam, and the UniForM Workbench [39] developed in Bremen.

All tools developed in the CoFI Tools group are made available to the community, after validation by the Tools group. A Web page for tools [25] describes on-going work and interests, giving access to available tools, and giving guidelines on how to propose a new tool.

9.5 Reactive Systems

The Reactive Systems Task Group is coordinated by Egidio Astesiano, Genova, and Heinrich Hussmann, Dresden.

An area of particular interest for applications is that of reactive, concurrent, distributed and real-time systems. There is considerable past work in algebraic specification that tackles systems of this kind, but nonetheless the application of Casl to such systems is speculative and preliminary in comparison with the rest of CoFI. The aim here is to propose and develop one or more extensions of Casl to deal with systems of this kind, and to study methods for developing software from such specifications. Extensions in three main categories are currently being considered:

- Combination of formalisms for concurrency (e.g. CCS, Petri nets, CSP) with Casl for handling classical (static) data structures;
- Formalisms built over Casl, where processes are treated as special dynamic data; and
- Approaches where Casl is used for coding, at the meta-level, some formalism for concurrency, as an aid to reasoning.

Since object-oriented methods have a dominant role in concrete developments, the group will address OO aspects insofar they are needed for reaching the above goals, but object orientation is not addressed as an independent topic in itself.

Work in this area begun only after the design of Casl was complete and so it is still in its early stages. Presently, the work is organized in two tracks: autonomous and coordinated extensions.

Autonomous Extensions: Proposals for such extensions are autonomously submitted to the Group; they are required to follow a suggested submission procedure, and are subject to approval [22]. Currently a number of proposals have been announced.

Coordinated Effort: It has been decided to start an effort centered around UML, the Unified Modeling Language. We are pursuing two tracks.

- The basic idea is to adopt CASL, or an extension of it, for *annotating UML*, thus possibly replacing OCL (work led by Heinrich Hussmann, Dresden).
- We have joined the activity of the free group *Precise UML*; the goal is to provide a formal/rigorous underpinning of UML, possibly exploiting CoFI-related techniques. A sketchy proposal for a general approach to the problem has been presented at a workshop at OOPSLA'98 [6]. Two other draft papers are available, one relating the ADT approach to UML [35] and the other proposing an underlying model for UML state machines [61].

9.6 External Relations

The External Relations Task Group is coordinated by Peter Mosses, Aarhus.

The design of CASL is based on a (critical) selection of constructs from existing languages, and it should be possible to translate specifications from other languages into (sublanguages or extensions) of CASL. The translation of a number of well-known algebraic specification languages to CASL at the level of specification in-the-small, namely Larch, ACT, OBJ3 [44], CafeOBJ [57], ASF+SDF [54], and HEP-theories, has been described [49]. Libraries and case studies that have been developed for these languages can be re-used in CASL, once the translations have been implemented. CoFI does not currently have adequate resources to study and implement translations of other languages into CASL, and must depend on attracting the interest and collaboration of those who have the necessary expertise.

The design of CASL has been sponsored by IFIP WG1.3 (on Foundations of System Specification), which also provided expert referees to review the proposed design in June 1997 [26,20]. The ongoing work in CoFI is of great interest to WG1.3, and Peter Mosses (chairman of WG1.3 since 1998) is responsible for liaison between CoFI WG and WG1.3.

All CoFI task groups welcome new participants. Please contact the coordinators via the CoFI web pages [18]. There is a moderated mailing list for each task group, with open subscription, administered by the Majordomo program (`majordomo@brics.dk`). All CoFI participants are requested to subscribe to a further mailing list, `cofi-list@brics.dk` (very low-volume, for major announcements only). All CoFI documents are available via the CoFI web pages [18].

Acknowledgements: This paper has been written by the coordinators of the CoFI Working Group. We acknowledge the significant contributions by members of the CoFI Task Groups to the design of Casl, in particular Hubert Baumeister, Didier Bert, Mark van den Brand, Maura Cerioli, Christine Choppy, Joseph Goguen, Radu Grosu, Magne Haveraaen, Anne Haxthausen, Kolyang, Till Mossakowski, Fernando Orejas, Olaf Owe, Gianna Reggio, Horst Reichel, Giuseppe Scollo, Frédéric Voisin, Martin Wirsing, Markus Roggenbach, Axel Schairer, and Bjarke Wedemeijer. In addition, we gratefully acknowledge the contribution of the IFIP WG 1.3 referees Hartmut Ehrig (coordinator), José Meseguer, Ugo Montanari, Fernando Orejas, Peter Padawitz, Francesco Parisi-Presicce, Martin Wirsing, and Uwe Wolter for their recommendations and suggestions for improving the initial proposed design.

References

[1] Robert Allen, David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, Volume 6, Number 3, pages 213-249, July 1997.

[2] David Aspinall. Type Systems for Modular Programs and Specifications. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh, 1997.

[3] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. INKA 5.0: a logic voyager. *Proc. 16th Intl. Conference on Automated Deduction*, Trento. *LNAI* volume 1632, pages 207–211. Springer, 1999. (For the INKA system see also `http://www.dfki.de/vse/systems/inka/`.)

[4] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. Towards an Evolutionary Formal Software Development Using Casl. In Christine Choppy, Didier Bert, and Peter Mosses (eds.): Recent Developments in Algebraic Development Techniques, 14th International Workshop, WADT'99, Chateau de Bonas, France. *LNCS* volume 1827, pages 73–88. Springer, 2000.

[5] Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner (eds.). *Algebraic Foundations of System Specification.* IFIP State-of-the-Art Reports, Springer 1999.

[6] Egidio Astesiano and Gianna Reggio. UML as Heterogeneous Multiview Notation: Strategies for a Formal Foundation. In *Proc. of OOPSLA'98 Workshop 'Formalizing UML. Why? How?'* Technical report, Universidade Nova de Lisboa, 1998.

[7] Jan Bergstra and Paul Klint. The discrete time ToolBus: A software coordination architecture. *Science of Computer Programming*, Volume 31, Number 2-3, pages 205–229. 1998.

[8] Michel Bidoit, Maria Victoria Cengarle and Rolf Hennicker. Proof Systems for Structured Specifications and Their Refinements. Chapter 11 of [5].

[9] Michel Bidoit, Hans-Jörg Kreowski, Pierre Lescanne, Fernando Orejas, and Donald Sannella (eds.). *Algebraic System Specification and Development: A Survey and Annotated Bibliography*, *LNCS* volume 501. Springer 1991.

[10] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Behavioural Encapsulation. Note L-28, in [18], 1996.

[11] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In *Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology (AMAST'98)*, *LNCS* volume 1548, pages 341–357. Springer, 1998.

[12] Rod Burstall and Joseph Goguen. Putting Theories Together to Make Specifications. *Proc. 5th Intl. Joint Conference on Artificial Intelligence*, Cambridge (USA), pages 1045–1058 (1977).

[13] Maura Cerioli, Martin Gogolla, Hélène Kirchner, Bernd Krieg-Brückner, Zhenyu Qian, and Markus Wolf (eds.). *Algebraic System Specification and Development: Survey and Annotated Bibliography.* 2nd edition, 1997. Monographs of the Bremen Institute of Safe Systems 3. ISBN 3-8265-4067-0. Shaker, 1998.

[14] Maura Cerioli, Anne Haxthausen, Bernd Krieg-Brückner and Till Mossakowski. Permissive Subsorted Partial Logic in CASL. In: Michael Johnson (ed.) *Proc. 6th Intl. Conference on Algebraic Methodology and Software Technology (AMAST'97)*, *LNCS* volume 1349, pages 91–107. Springer, 1997.

[15] Maura Cerioli, Till Mossakowski and Horst Reichel. From Total Equational to Partial First-Order Logic. Chapter 3 of [5].

[16] Ingo Claßen. Revised ACT ONE: Categorical Constructions for an Algebraic Specification Language. *Proc. Workshop on Categorical Methods in Computer Science with Aspects from Topology*, Berlin. *LNCS* volume 393, pages 124–141. Springer, 1989.

[17] Ingo Claßen, Hartmut Ehrig and Dietmar Wolz. *Algebraic Specification Techniques and Tools for Software Development.* AMAST Series in Computing, World Scientific (1993).

[18] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by WWW at `http://www.brics.dk/Projects/CoFI`.

[19] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Design Proposal. Documents/CASL/Proposal, in [18], May 1997.

[20] CoFI Language Design Task Group. Response to the Referee Report on Casl. Documents/CASL/RefereeResponse, in [18], August 1997.

[21] CoFI Language Design Task Group. Casl – The CoFI Algebraic Specification Language – Summary (version 1.0). Documents/CASL/Summary, in [18], October 1998 (adjusted July 1999).

[22] CoFI Reactive Systems Task Group. Pattern for proposals of Casl extensions for Reactive Systems. `Reactive/Pattern/`, in [18].

[23] CoFI Semantics Task Group. Casl – The CoFI Algebraic Specification Language (version 0.97) – Semantics. Note S-6, in [18], July 1997.

[24] CoFI Semantics Task Group. Casl – The CoFI Algebraic Specification Language – Semantics (version 1.0). Note S-9, in [18], 2000.

[25] CoFI Tools Task Group. The CoFI Tools Group Home Page. `http://www.loria.fr/~hkirchne/CoFI/Tools/index.html`

[26] IFIP WG 1.3. Referee Report on Casl. Documents/CASL/RefereeReport, in [18], June 1997.

[27] Werner Fey. Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language. Ph.D. thesis, Report 88/26, Technische Universität Berlin (1988).

[28] GNU project. `htpp://www.gnu.org/`.

[29] Joseph Goguen. Stretching first order equational logic: proofs with partiality, subtypes and retracts. Available from `http://www-cse.ucsd.edu/users/goguen/pubs/`. Submitted for publication (1998).

[30] Joseph Goguen and Rod Burstall. Institutions: abstract model theory for specification and programming. *Journal of the Assoc. for Computing Machinery* 39:95–146 (1992).

[31] Joseph Goguen and José Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 105, pages 217–273 (1992).

[32] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud. Introducing OBJ. In: *Software Engineering with OBJ: Algebraic Specification in Action* (J. Goguen and G. Malcolm, eds.). Kluwer Academic (2000).

[33] John Guttag and Jim Horning. Report on the Larch shared language. *Science of Computer Programming* 6:103–134 (1986).

[34] John Guttag and Jim Horning. *Larch: Languages and Tools for Formal Specification.* Springer (1993).

[35] Heinrich Hußmann, Maura Cerioli, Gianna Reggio, and Françoise Tort. Abstract Data Types and UML Models. Technical report, DISI – Università di Genova, DISI-TR-99-15, 1999.

[36] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1986.

[37] Einar W. Karlsen. Interoperability of Casl tools using CORBA. Note T-5, in [18], October 1997.

[38] Hélène Kirchner and Christophe Ringeissen. Executing Casl Equational Specifications with the ELAN Rewrite Engine. Note T-9, in [18], October 1999.

[39] Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, and Alexander Baer. The UniForM Workbench, a Universal Development Environment for Formal Methods. In: J. M. Wing, J. Woodcock, and J. Davies (eds.): FM'99, Formal Methods. Proceedings, Vol. II. *LNCS* Volume 1709, pages 1186-1205. Springer, 1999. (for the UniForM Workbench tools see also `http://www.informatik.uni-bremen.de/~uniform`)

[40] Bernd Krieg-Brückner. Seven Years of COMPASS. In *11th Workshop on Specification of Abstract Data Types, Joint with the 8th* COMPASS *Workshop, Oslo*, *LNCS* volume 1130, pages 1–13. Springer, 1996.

[41] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types.* Wiley, 1996.

[42] José Meseguer. General Logics. In H.-D. Ebbinghaus, J. Fernández-Prida, M. Garrido, D. Lascar, and M. Rodríguez Artalejo (eds.) *Logic Colloquium '87*, pages 275–329. North-Holland, 1989.

[43] Pierre-Etienne Moreau and Hélène Kirchner. A compiler for rewrite programs in associative-commutative theories. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, *LNCS* volume 1490, pages 230–249. Springer, September 1998. Report LORIA 98-R-226.

[44] Till Mossakowski. Translating OBJ3 to Casl: the institution level. *Recent Trends in Algebraic Development Techniques: Selected Papers from WADT'98*, Lisbon. *LNCS* volume 1589, pages 198–214. Springer, 1999.

[45] Till Mossakowski. Sublanguages of Casl. Note L-7, in [18], December 1997. Update in [49].

[46] Till Mossakowski. Two 'Functional Programming' Sublanguages of Casl. Note L-9, in [18], March 1998.

[47] Till Mossakowski. Specifications in an arbitrary institution with symbols. In Christine Choppy, Didier Bert, and Peter Mosses (eds.): Recent Developments in Algebraic Development Techniques, 14th International Workshop, WADT'99, Chateau de Bonas, France. *LNCS* volume 1827, pages 252–270. Springer, 2000.

[48] Till Mossakowski. Casl: From Semantics to Tools. In S. Graf (eds.) *TACAS 2000*, *LNCS* volume 1785, pages 93–108. Springer, 2000.

[49] Till Mossakowski. Relating Casl with Other Specification Languages: the Institution Level. Theoretical Computer Science, this volume.

[50] Till Mossakowski, Anne Haxthausen, and Bernd Krieg-Brückner. Subsorted Partial Higher-Order Logic as an Extension of CASL. In Christine Choppy, Didier Bert, and Peter Mosses (eds.): Recent Developments in Algebraic Development Techniques, 14th International Workshop, WADT'99, Chateau de Bonas, France. *LNCS* volume 1827, pages 126–145. Springer, 2000.

[51] Till Mossakowski, Kolyang, and Bernd Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In *12th Workshop on Algebraic Development Techniques, Tarquinia*, *LNCS* volume 1376, pages 333–348. Springer, 1998. (For the Bremen CoFI Tools see `http://www.tzi.de/cofi`.)

[52] Till Mossakowski, and Markus Roggenbach. The Datatypes REAL and COMPLEX in CASL. Note M-7, in [18], April 1999.

[53] Peter D. Mosses. CoFI: The Common Framework Initiative for algebraic specification. *Bull. EATCS*, (59):127–132, June 1996.

[54] Peter D. Mosses. CASL for ASF+SDF users. In *ASF+SDF'97, 2nd Intl. Workshop on the Theory and Practice of Algebraic Specifications*, volume ASFSDF-97 of *Electronic Workshops in Computing*. Springer, 1997. `http://www.ewic.org.uk/ewic/workshop/view.cfm/ASFSDF-97`.

[55] Peter D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97: Theory and Practice of Software Development*, *LNCS* volume 1214, pages 115–137. Springer, 1997.

[56] Peter D. Mosses. Formatting CASL specifications using LaTeX. Note C-2, in [18], June 1998.

[57] Peter D. Mosses. CASL for CafeOBJ Users. In: Kokichi Futatsugi and others (eds.): Cafe on Networks: An Industrial-Strength Algebraic Formal Method. Elsevier (to appear).

[58] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford Univ. Press, 1990.

[59] Larry Paulson. *Isabelle: A Generic Theorem Prover*. *LNCS* volume 828. Springer, 1994.

[60] Wolfgang Reif. The KIV-approach to Software Verification. In *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, *LNCS* volume 1009, pages 339–368. Springer, 1995. (For the KIV system see also `http://www.informatik.uni-ulm.de/pm/kiv/kiv.html`.)

[61] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hußmann. Making Precise UML Active Classes Modeled by State Charts. Technical report, DISI – Università di Genova, DISI-TR-99-14, 1999.

[62] Markus Roggenbach, Lutz Schröder, and Till Mossakowski. Specifying Real Numbers in CASL. In Christine Choppy, Didier Bert and Peter Mosses (eds.): Recent Developments in Algebraic Development Techniques, 14th International Workshop, WADT'99, Chateau de Bonas, France. *LNCS* volume 1827, pages 146–161. Springer, 2000.

[63] Markus Roggenbach, and Till Mossakowski. Rules of Methodology. Note M-6, in [18], 2000.

[64] Markus Roggenbach, and Till Mossakowski. Basic Datatypes in CASL. Note L-12, in [18], March 2000.

[65] Donald Sannella, Stefan Sokołowski and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29, pages 689–736 (1992).

[66] Donald Sannella and Andrzej Tarlecki. Extended ML: an institution-independent framework for formal program development. *Proc. Intl. Workshop on Category Theory and Computer Programming*, Guildford. *LNCS* volume 240, pages 364–389. Springer, 1986.

[67] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76, pages 165–210, 1988.

[68] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* 9, pages 229–269 (1997).

[69] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development.* Cambridge Univ. Press, to appear.

[70] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. Proc. 1983 Int'l. Conf. on Foundations of Computation Theory, Borgholm. *LNCS* Volume 158, pages 413-427. Springer, 1983.

[71] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Data type specification: parameterization and the power of specification techniques. ACM TOPLAS, Volume 4, pages 711–732, 1982.

[72] Christophe Tronche. The Cachan Parser for CASL. `http://www.lsv.ens-cachan.fr/~tronche/cofi/`

[73] Mark G.J. van den Brand. The Amsterdam Parser for CASL. `http://adam.wins.uva.nl/~markvdb/cofi/casl.html`

[74] Mark G.J. van den Brand, Hayco A. de Jong, Paul Klint and Pieter A. Olivier. Efficient Annotated Terms. *Software-Practice and Experience 30*, pp 259–291, Wiley, 2000. (For the ATerm library see `http://www.cwi.nl/projects/MetaEnv/aterm/`.)

[75] Mark G.J. van den Brand and Jeroen Scheerder. Development of Parsing Tools for CASL Using Generic Language Technology. In Christine Choppy, Didier Bert, and Peter Mosses (eds.): Recent Developments in Algebraic Development Techniques, 14th International Workshop, WADT'99, Chateau de Bonas, France. *LNCS* volume 1827, pages 89–105. Springer, 2000.

[76] Martin Wirsing. Algebraic specification. In: J. van Leeuwen, (ed.) *Handbook of Theoretical Computer Science*, Volume B, pages 675-788. North-Holland, 1990.