2

Underlying Concepts

Casl is based on standard concepts of algebraic specification.

This chapter reviews the main concepts of algebraic specification. It briefly explains and illustrates standard terminology regarding specification language constructs and models of specifications (i.e., algebras), and indicates the differences between basic, structured, and architectural specifications.

The focus here is on concepts that are relevant to Casl, and which will be needed in later chapters. For comprehensive presentations of concepts and results concerning algebraic specification, see [3, 10, 16, 27, 34, 35, 37]; for an overview of the design of Casl, see [2]; and for full details of Casl, see the Casl Reference Manual [20].

The reader is assumed to be familiar with basic mathematical notions (sets, relations, and total and partial functions) and with the use of logical formulas as axioms.

2.1 Basic Specifications

A basic specification declares symbols, and gives axioms and constraints.

A *basic specification* in an algebraic specification language generally consists of a set of *declarations* of *symbols*, and a set of *axioms* and *constraints*, which restrict the *interpretations* of the declared symbols. Casl allows basic specifications to include also items which simultaneously declare symbols and restrict their interpretations.

© IFIP International Federation for Information Processing 2004

The semantics of a basic specification is a signature and a class of models.

The *meaning* or *semantics* of a basic specification *SP* generally has two parts:

- a *signature* Σ, corresponding to the *symbols* introduced by the specification, and
- a *class of* Σ*-models*,[1] corresponding to those *interpretations* of the signature Σ that *satisfy* the *axioms* and *constraints* of the specification.

When a model M satisfies a specification SP, we write $M \models SP$ and say that M is a *model of SP*. (Formalizing this within the theory of so-called *institutions* involves categorical structure on the set of signatures and on the class of models, and a natural condition on the *satisfaction relation.* We need not bother with the details here – but see however the concept of a signature morphism in Sect. 2.2.) A specification is said to be *consistent* when its class of models is non-empty, and otherwise *inconsistent.*

CASL specifications may declare sorts, subsorts, operations, and predicates.

A CASL *signature* represents declarations of sorts, subsorts, operations, and predicates. The signature is called *many-sorted* when there are no subsort declarations, and otherwise *subsorted*; it is called *algebraic* when there are no predicate declarations.

Sorts are interpreted as carrier sets.

A *sort* is a symbol which is interpreted as a set, called a *carrier set.* The elements of a carrier set are generally abstract representations of the data processed by software: numbers, characters, lists, etc. Thus a sort declared by a specification corresponds to a type in a programming language. Sort symbols are usually chosen to be strongly suggestive of their intended interpretations, e.g., *Int* for a sort to be interpreted as the set of integers, *List* for a set of lists. CASL allows also *compound sort* symbols, such as *List*[*Int*] for lists of integers.

[1] Readers who are not interested in foundational aspects may treat the word 'class' as a synonym for 'set'. In general, the models of an algebraic specification in CASL constitute a *proper class*, because there is no restriction on the elements of the carrier sets.

Subsorts declarations are interpreted as embeddings.

A sort may be declared to be a *subsort* or a *supersort* of other sorts. The subsort relation between two sorts could be interpreted as *set inclusion*. Its interpretation in CASL is however more general: it is interpreted as an *embedding*, i.e., a 1-1 function from the carrier set of the subsort to that of the supersort. For example, if *ASCII* is specified to be a subsort of *ISO_Latin1* in CASL, the carrier set for *ASCII* could be simply a subset of that for *ISO_Latin1*. If *Char* were to be declared as a subsort of *String*, however, the carrier sets for *Char* and *String* could be disjoint, with the embedding mapping each character to the corresponding single-character string. (See also the concept of overloading, below.)

Operations may be declared as total or partial.

An *operation* symbol consists of the name of the operation together with its *profile*, which indicates the number and sorts of the arguments, and the *result sort*. In CASL, a declared operation symbol is interpreted as either a *total* or a *partial function* from the Cartesian product of the carrier sets of the argument sorts to the carrier set of the result sort; the subset of the argument tuples for which the result of a function is defined is called its *domain of definition*. The declaration indicates whether the function is total or partial.[2] For example, integer addition would be declared as total, but integer division as partial. The result of applying an operation is undefined whenever any of its arguments is undefined (regardless of whether the operation itself is total or partial).

When there are no arguments, the operation is called a *constant*. A constant is interpreted simply as an element of the result sort.

Predicates are different from boolean-valued operations.

A *predicate* symbol consists of the name of the predicate together with its *profile*, which indicates the number and sorts of the arguments but no result sort: predicates are syntactically different from boolean-valued operations, and are used to form atomic formulas rather than terms. In CASL, a declared predicate symbol is interpreted as a relation on (i.e., a subset of) the Cartesian product of the carrier sets of the argument sorts. An application of a predicate is said to *hold* when the tuple of arguments is in the relation. For example, a

[2] A partial function might just happen to be everywhere defined, of course.

symbol ‘<’ to be interpreted as the less-than relation could be declared as a binary predicate on integers.

An application of a predicate simply fails to hold when any of its arguments is undefined: there is no undefinedness about holding or not. This allows the logic to remain two-valued, and the logical connectives to have their familiar interpretations.

In contrast, the result of evaluating an application of even a total boolean-valued operation could be true, false, or undefined: the last case arises when any argument of the application is undefined. Thus boolean-valued operations corresponding to logical connectives (conjunction, implication, etc.) have to take account of undefinedness, which leads to a three-valued logic.

A further significant difference between predicates and boolean-valued operations shows up in connection with the concept of initiality, see Sect. 2.2. (Predicates of two-valued logic can be represented accurately by *partial* operations with a *single-valued* result sort, holding being represented by definedness.)

Operation symbols and predicate symbols may be overloaded.

An operation or predicate name can be declared with different profiles in the same specification. This is called *overloading*. For example, the constant ‘*empty*’ could be overloaded, being interpreted as (unrelated) elements of the sorts *List* and *Set*, according to the context of its use. Similarly, a predicate name such as ‘<’ could be overloaded on unrelated sorts such as *Char* and *Int*.

In CASL, overloading is required to be *compatible* with embeddings between subsorts. For example, the sort *Nat*, interpreted as the set of natural numbers, might be a subsort of *Int*, interpreted as the set of all integers; then when the operation name ‘+’ and the predicate name ‘<’ are declared both on *Nat* and on *Int*, their interpretations are required to be such that it makes no difference whether the embedding from *Nat* to *Int* is applied to the arguments or to the result of the operation, and whether it is applied to the arguments of the predicate or not.

Axioms are formulas of first-order logic.

The interpretation of quantification (universal, existential, and unique-existential) and of the usual logical connectives (negation, conjunction, disjunction, implication, and equivalence) in CASL axioms is completely standard. Variables in formulas range over the carrier sets of specified sorts.

Apart from the usual predicate applications, the atomic formulas in Casl axioms are equations (strong or existential), definedness assertions, and subsort membership assertions. An *existential* equation holds when the values of its terms are defined and equal; a *strong* equation holds moreover when the values of the terms are both undefined.

Regardless of whether the values of the terms occurring in an axiom are defined, the axiom either holds or it does not hold in a particular model: the logic is two-valued, there is no "maybe" or undefinedness about the holding of axioms. Recall that when the value of any argument term is undefined, an application of a predicate never holds; similarly, definedness and subsort membership assertions never hold when their arguments are undefined.

Sort generation constraints eliminate 'junk' from specific carrier sets.

In general, the carrier sets of the models of a specification may contain 'junk' elements, i.e., elements which cannot be obtained by any composition of the operations declared by the signature of the specification.

A *sort generation constraint* in Casl concerns specific sorts and operations, and is satisfied in a model when no elements of the indicated carrier sets are junk with respect to the indicated operations – i.e., all the elements of those sets can be obtained by consecutively applying those operations to elements of the carrier sets of the remaining sorts. For example, the carrier set for the sort *Container* might be constrained to be generated from that for the sort *Elem* by the following operations:

- a constant '*empty*' of sort *Container*, and
- a binary operation '*insert*' with argument sorts *Elem* and *Container*, and result sort *Container*.

This constraint would ensure that the only elements of the *Container* carrier are those obtained by a finite number of successive applications of the *insert* operation to elements of sort *Elem*, starting with the *empty* value of sort *Container*.

2.2 Structured Specifications

Structured specifications are formed from basic specifications, *references* to *named specifications*, and *instantiations* of *generic specifications*, using various constructs for composing specifications.

The semantics of a structured specification is simply a signature and a class of models.

The semantics of a structured specification is of the same form as that of a basic specification: a signature, together with a class of models. Thus the structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style. (Specification of the *architecture* of implementations is addressed in Sect. 2.3.) The symbols in the signature are called the *exported* symbols of the specification.

The interpretation of structured specification constructs involves mappings between signatures Σ, called *signature morphisms*, and corresponding mappings between models M, called *reducts along morphisms*. In CASL, a signature morphism σ from Σ to Σ' consists of a mapping which gives:

- for each sort of Σ, a corresponding sort of Σ', preserving any subsort relationships, and
- for each operation or predicate symbol whose profile has sorts in Σ, a corresponding symbol in Σ' whose profile has the corresponding sorts, preserving any overloading between symbols whose profiles are related by subsorting. A partial operation may be mapped to a total operation, but not vice versa.

Let M' be any Σ'-model. We can define its *reduct* along the signature morphism σ to be the Σ-model M obtained as follows: each symbol of Σ is interpreted in M in exactly the same way as the corresponding symbol in Σ' is interpreted in M'. Conversely, given M, a model M' is said to be an *expansion* of M when the reduct of M' is M.

Suppose that a specification SP has signature Σ and SP' has signature Σ'. A signature morphism σ from Σ to Σ' is said to be a *specification morphism* from SP to SP' when the reduct along σ of each model of SP' is a model of SP.

For two Σ-models M_1, M_2, a (weak) *homomorphism* from M_1 to M_2 maps the elements of the carrier sets of M_1 to the elements of the corresponding carrier sets of M_2, preserving the embeddings between subsorts, the values (and definedness) of operations, and the holding of predicates. A homomorphism is an *isomorphism* when it has an inverse homomorphism.

A model M is *initial* in a class of Σ-models if there is a unique homomorphism from M to each model in the class. When a class of models has an initial model (which need not be the case in CASL) it is unique, up to isomorphism.

With reference to the above concepts of signature morphism, model reduct, and homomorphism, we can now proceed to explain the constructs involved with structured specifications.

A translation merely renames symbols.

Translating a sort symbol requires translating the profiles of all operation and predicate symbols involving that sort; translating an operation or predicate symbol has to respect overloading between symbols whose profiles are related by subsorting. The translation of sort, operation, and predicate names in CASL determines a signature morphism σ mapping the signature Σ of a specification SP onto a new signature Σ'. The models of the translation specification are all those models interpreting Σ' whose reducts along σ are models of SP.

Hiding symbols removes parts of models.

Hiding a sort symbol implies hiding also all operation and predicate symbols whose profiles involve that sort; hiding an operation or predicate symbol, however, does not have further implications. Hiding a set of symbols that occur in the signature Σ of a specification SP to give a subsignature Σ' determines a signature morphism σ which simply includes Σ' in Σ. The models of the hiding specification are the reducts of the models of SP along σ.

For example, the operation *suc* might be introduced purely to facilitate the specification of the natural numbers, with sort *Nat*, constants *0* and *1*, and the usual arithmetic operations. Hiding *suc* removes the interpretation of *suc* from the models of the specification[3] but leaves the carrier set for *Nat* unchanged.

Union of specifications identifies common symbols.

The signature of a union of specifications SP_1, SP_2 is simply the union of their respective signatures Σ_1, Σ_2. The models of the union are those models of the union signature whose reducts to Σ_1 and Σ_2 along the signature inclusions satisfy SP_1 and SP_2 respectively. Thus each symbol that the two signatures have in common has a single interpretation in any model of the union specification. This is known as the 'same name, same thing' principle.

[3] The successor of a number can of course still be obtained, using addition and *1*.

Extension of specifications identifies common symbols too.

The signature of the extension of a specification *SP* by further specification items (declarations, axioms, and constraints) is simply the extension of the signature Σ of *SP* with the symbols of the new declarations. The models of the extension are those models of the extended signature which satisfy the axioms and constraints specified by the extension and whose reducts to Σ satisfy *SP*. If the extension redeclares a symbol of *SP*, there is still only one occurrence of that symbol in the signature of the extension, and hence only one interpretation of it – again the 'same name, same thing' principle.

In CASL, unions, extensions, and other kinds of structured specification can be formed from specification fragments that determine only signature *extensions*, not necessarily complete signatures.

Free specifications restrict models to being free, with initiality as a special case.

When a specification is freely extended by additional specification items, the interpretations of the additional declarations are required to satisfy the axioms, but nothing more: that is, properties that are not consequences of the axioms should not hold. In particular, the domains of definition of partial operations – and the sets of arguments for which predicates hold – are as small as possible. The carriers for the original sorts are left unchanged; any new carriers are no larger than is required to provide interpretations for the operations, without unnecessary junk elements. This restriction of the models is referred to as a *freeness constraint*. In the degenerate case where the specification being enriched is empty, the models of the free extension are just the initial models.

The difference between predicates and boolean-valued operations is particularly apparent in free specifications: with predicates, it is only required to specify when they hold, since not holding is the default; with boolean-valued operations, however, the true and false values are treated symmetrically, and it is necessary to specify both cases, since neither is the default.

Generic specifications have parameters, and have to be instantiated when referenced.

A named specification may declare some *parameters*, the union of which is extended by a *body*; it is then called *generic*. The purpose of a generic specification is to reuse the body in different contexts; hence a reference to a generic specification should *instantiate* it by providing, for each parameter, an *argument specification* together with a *fitting morphism* from the parameter to the argument specification. Fitting may also be achieved by use of named *views* between the parameter and argument specifications. The instantiation of the generic specification gives the union of the arguments, together with the translation of the generic specification by an expansion of the fitting morphism. This corresponds to a so-called push-out construction – taking into account any explicit *imports* of the generic specification.

2.3 Architectural Specifications

The semantics of an architectural specification reflects its modular structure.

The intention with architectural specifications is primarily to impose structure on implementations, expressing their *composition* from component units – and thereby also a *decomposition* of the task of developing such implementations, from requirements specifications. This is in contrast to the structured specifications considered in Sect. 2.2, where the specified models have no more structure than do those of the basic specifications considered in Sect. 2.1.

Architectural specifications involve the notions of persistent function and conservative extension.

A function F mapping Σ-models to Σ'-models, where the signature Σ' extends Σ, is said to be *persistent* when for each Σ-model M, the reduct of $F(M)$ to a Σ-model is exactly M.

A specification extension SP' of SP is said to be *conservative* when each model of SP can be expanded to a model of SP'.

A persistent function mapping models of SP to models of SP' exists only if SP' is a conservative extension of SP.

2.4 Libraries of Specifications

The semantics of a library of specifications is a mapping from the names of the specifications to their semantics.

The specification of a library gives also a library name, and determines a version number.