8 Specification Languages

Donald Sannella[1] and Martin Wirsing[2]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland; dts@dcs.ed.ac.uk, http://www.dcs.ed.ac.uk/~dts/
[2] Institut für Informatik, Ludwig-Maximilians-Universität München, München, Germany; wirsing@informatik.uni-muenchen.de, http://www.pst.informatik.uni-muenchen.de/personen/wirsing/

8.1 Introduction

The basic components of any specification language include: constructs for specifying the properties of individual program components such as types and functions; structuring mechanisms for building large specifications in a modular fashion; a description of the semantics of the language; mechanisms for performing proofs of properties of specifications; a notion of refinement of specifications; and a way of relating specifications to programs written in a programming language. These topics are discussed in other chapters of this book, and in each case there are various alternatives to choose from. To take a simple example, the properties of functions may be specified using equations or using first-order (or even higher-order) formulas as axioms.

A specification language is a commitment to a compatible combination of these choices. It is a commitment because the syntax of a specification language determines once and for all what can be expressed and what cannot be expressed. For example, the syntax of formulas determines how the properties of functions are specified; similarly, a construct for hiding types and/or functions may or may not be included. The choices must be compatible in various respects. On one hand, certain combinations of choices simply make no sense; for instance, initial semantics requires the use of axioms no more complex than conditional equations. More subtly, the inclusion of certain structuring mechanisms complicates the process of proving properties of specifications and so these might be omitted or restricted in a language in which proofs are of primary importance.

There is no single best combination of choices because this depends on many factors including the intended context of use and the range of applications to be addressed. Consequently, this chapter will consider the rationale for choosing between the various alternatives available. The technical details of these alternatives are given in other chapters for the most part and will not be repeated here.

The next section is devoted to a brief overview of some existing specification languages. Section 8.3, which forms the bulk of the chapter, discusses the

various design decisions that would confront the designer of a new specification language. The final short section speculates on some likely future trends. As in other chapters of this book, we focus on property-oriented specifications, although many of the issues discussed relate to specification languages in general.

8.2 Existing specification languages

In this section we give a brief summary of the features of a range of existing specification languages, with a small example of a specification in each language. Further small examples of specifications written in these languages will appear in the next section. Note that these examples are chosen to illustrate language features rather than for their intrinsic interest. See [Wir95] for a more comprehensive survey with historical remarks.

Clear and OBJ3. Clear [BG77,BG80], the first algebraic specification language, had considerable influence on the design of many other languages. Features of Clear include: the use of specification-building operations for constructing specifications in a modular way, with care taken to respect shared sub-specifications, and most of the particular operations used in other languages (see Chapter 6); explicit pushout-based parameterisation (Chapter 6); free generating constraints (Chapter 5); institution independent theory-level semantics (Chapters 4 and 5); a structured proof system [SB83] (Chapter 11); and a notion of implementation of parameterised specifications [SW82] (Chapter 7). OBJ3 [GWM+92] is an executable specification language, also known as an "ultra-high-level" programming language, that can be seen as an implementation of Clear (using associative-commutative rewriting, see Chapter 9) for the institution of order-sorted conditional equational logic. Order-sorted logic is used in order to cope smoothly with partial functions, errors and subsort polymorphism; see Chapter 3. OBJ3 has several variants of Clear's enrich specification-building operation, including so-called "protecting importation" which ensures that the enrichment is persistent. It supports a very flexible "mixfix" notation with operators such as $\mathit{if\_then\_else\_fi}\colon \mathit{expr} \times \mathit{stmt} \times \mathit{stmt} \to \mathit{stmt}$. Two new object-oriented specification languages are based on OBJ3: Maude [Mes93] extends OBJ3 by a notion of objects and by concurrent rewriting for describing the dynamic behaviour of objects; CafeOBJ [DF98] is similar but focuses additionally on behavioural specifications (see Chapter 5).

The following simple example in OBJ3 demonstrates the use of subsorts. The operation `tail` can only be applied to values of sort `NeList` (non-empty lists), which is specified as a subsort of `List` and a supersort of the sort `Elt` of list elements (from the "interface theory" `TRIV`).

```
obj LIST[X :: TRIV] is
  sorts List NeList .
  op nil : -> List .
```

```
  subsorts Elt < NeList < List .
  op __ : List List -> List [assoc id: nil] .
  op __ : NeList List -> NeList .
  op __ : NeList NeList -> NeList .
  protecting NAT .
  op |_| : List -> Nat .
  eq | nil | = 0 .
  var E : Elt .   var L : List .
  eq | E L | = 1 + | L | .
  op tail_ : NeList -> List .
  var E : Elt .   var L : List .
  eq tail E L = L .
endo
```

ACT ONE and ACT TWO. ACT [CEW93] is an approach to formal software development that includes a language called ACT ONE [Cla89] for writing parameterised specifications, called "types", with conditional equational axioms and initial constraints, and an extension called ACT TWO [Fey88] for writing specified modules. ACT ONE has a pure initial algebra semantics (Chapter 5) where every type is parameterised (non-parameterised types are considered as a degenerate case) and denotes a free functor. Thus it provides only simple specification-building operators (free extension, union, renaming and pushout-based instantiation) but no operation for hiding. A module in ACT TWO consists of four specifications:

Parameter: This describes parameters that are common to the entire module or modular system in which the module appears, e.g., the underlying character set.

Import interface: This describes the sorts and operations that the module requires to be supplied by other modules.

Export interface: This describes the sorts and operations that the module supplies for use by other modules.

Body: This defines the construction of the exported components in terms of the imported components. This construction may involve auxiliary operations that are not exported.

These four specifications are written in ACT ONE extended by permitting first-order axioms (except in the body part). Module-building operations (composition, union, instantiation and renaming) are module-level analogues to specification-building operations. Both ACT ONE and ACT TWO have both a presentation-level and a model-level semantics. For each language these are described separately and are then shown to be compatible.

The following example is an ACT ONE parameterised type of lists. The type `ELEM` serves as the formal parameter for the type `LIST`, meaning that the body of `LIST` is interpreted as an extension of `ELEM`. `List(D)` is a so-called "sort with structured name", expressing the dependency of the sort `List` on the parameter sort `D`.

```
parameter ELEM is
  sorts D
endpar

type LIST[ELEM] is
  extend
    union NAT, BOOL endunion
  by
    sorts
      List(D)
    constructors
      nil: -> List(D);
      cons: D, List(D) -> List(D);
    functions
      vars x:D, l:List(D);
      func length: List(D) -> Nat;
        length(nil) = zero;
        length(cons(x,l)) = succ(length(l));
      func isempty: List(D) -> Bool;
        isempty(nil) = true;
        isempty(cons(x,l)) = false;
  endext
endtype
```

The operation of instantiating `LIST` with a specification `NAT` of natural numbers is called "actualization", and is written as follows:

```
type NATLIST is
  actualize
    LIST by NAT using sortnames Nat for D
  endact
endtype
```

The instantiation of the sort `D` causes the resulting type `NATLIST` to contain a sort with the name `List(Nat)`.

ASL. ASL [SW83,Wir86,ST88a] is a kernel language that is intended to provide a sound semantic basis for defining more user-friendly high-level specification languages (e.g., Pluss [BGM89,Bid89] and the meta-language used in [ABB+86]). It consists of a small number of simple but powerful specification-building operations with an institution-independent loose model-class semantics (Chapters 4 and 5). Generating constraints are incorporated via a specification-building operation, called reachable. A specification-building operation called abstract (and a special case of abstract called behaviour) provides observational abstraction (Chapter 5). The absence of initiality constraints means that there is no reason to restrict to the use of conditional equational axioms, so more expressive logics – typically first-order logic with equality – are used. ASL provides explicit λ-calculus-based higher-order parameterisation. [SST92] describes an extension of ASL called ASL+ that also

supports specification of (possibly higher-order) parameterised algebras, and distinguishes between these and parameterised specifications, with a formal system for proving satisfaction, cf. [Asp97]. The expressive power of the language makes it possible to model refinement of specifications (Chapter 7) as model class inclusion; more elaborate notions of implementation of ASL specifications are studied in [ST88b]. A system for performing structured proofs of properties of specifications and of correctness of implementation steps (Chapter 11) is described in [Far92], and complete proof systems for subsets of ASL are given in [Wir93,Cen94,HWB97,Hen97].

The following is an ASL specification of bags of natural numbers with a counting function based on a predefined specification of natural numbers (`NAT`) that is assumed to be monomorphic. The use of reachable ensures that only term-generated models are admissible. The axioms ensure that the data structure of finite multisets of natural numbers is (up to isomorphism) the only model of this specification.

```
BAGcount =
  reachable
    enrich NAT by
      sorts  Bag
      opns   empty : Bag
             cons: Nat, Bag -> Bag
             count: Nat, Bag -> Nat
      axioms cons(x, cons(y, b)) = cons(y, cons(x, b))
             count(x,empty) = zero
             count(x, cons(x,b)) = succ(count(x, b))
             x≠y ⇒ count(x, cons(y,b)) = count(x, b)
  on {Bag}
```

Using the behaviour operator, the class of models is enlarged to include, e.g., the data structure of finite lists of natural numbers, which is the initial model of `BEHAVIOURAL_SET`; the models of `BAGcount` are its final models.

```
BEHAVIOURAL_SET = behaviour BAGcount wrt {Nat}
```

Larch. Larch [GH86,GH93] is a family of specification languages. Each Larch specification has components written in two languages: one designed for a specific programming language, the so-called Larch interface language, and another common to all programming languages, the so-called Larch shared language LSL. LSL is a loose algebraic specification language with equational axioms, generating constraints, and simple structuring operators (rename, combine, and implicit parameterisation) but no operation for hiding, and a theory semantics based on first-order logic with equality. There is a construct for observational abstraction called partitioned by which has the status of a constraint, and specifications, called "traits", may contain assertions about intended theorems using the keyword implies. The LP theorem prover supports proof of properties of LSL traits. Larch interface languages have been

designed for CLU, Ada, C, Modula-3, Smalltalk, and C++. Each of these is an extension of a programming language with annotations that describe program properties expressed via abstract concepts from the problem domain that are defined in LSL traits.

The following is an LSL trait defining bags based on a predefined trait of natural numbers (`NAT`). Sorts are declared implicitly by their appearance in the signature, and the parameters in the declaration of a trait may contain names of sorts or function symbols. Parameters are used merely to facilitate renaming, so the appearance or non-appearance of parameters has no semantic consequences. Axioms are given following the keyword asserts. The generated by assertion amounts to a reachability constraint for `Bag`, while the partitioned by assertion says that values of sort `Bag` that cannot be distinguished using `count` are equal. The implies clause states intended consequences of the trait, where `converts count` says that the definition of `count` is sufficiently complete.

```
BAGcount (E, Bag): trait
  includes NAT
  introduces
    empty: → Bag
    cons: E, Bag → Bag
    count: E, Bag → Nat
  asserts
    Bag generated by empty, cons
    Bag partitioned by count
    ∀ b: Bag, x,y: E
      count(x,empty) == zero;
      count(x,cons(y,b)) == if x=y then succ(count(x,b))
                                   else count(x,b)
  implies
    converts count
    ∀ x: E, b: Bag
      cons(x,b) ≠ empty
```

A partial instantiation of `BAGcount` with natural numbers for the element sort `E` is achieved by defining a new trait that includes `NAT` and `BAGcount` appropriately renamed:

```
NatBAGcount: trait
  includes
    NAT, BAGcount (Nat, Bag)
```

Note that `BAGcount (Nat, Bag)` is just an abbreviation for the renaming `BAGcount (Nat for E)`.

Extended ML. Extended ML (EML) [San91,ST91,KS98] is a framework for the formal development of modular programs in the Standard ML (SML)

functional programming language [Pau96] from specifications of their required behaviour. The EML language is a "wide-spectrum" language that is based on a large subset of SML, excluding mainly references and input/output. EML thus inherits SML's higher-order polymorphic type system and its facilities for building hierarchically-structured and parameterised program modules with explicit interfaces. EML extends this subset of SML by permitting axioms in module interfaces, for specifying required properties of module components, and in place of code in module bodies, for describing functions in a non-algorithmic way prior to their implementation as SML code. Axioms are written in a language of higher-order logic with equality, with additional features to deal with run-time exceptions and non-terminating functions. Correctness of module bodies with respect to their interfaces is required only up to behavioural equivalence. The semantics of EML [KST94,KST97] is based directly on the static and dynamic semantics of SML [MTH90], extended with a loose model-class semantics to give meaning to the specification constructs. The relationship between EML and SML is thus a formal one based on semantics in contrast to the rather informal relationship between specification language and programming language in the case of Larch. Spectral [KS91] can be seen as an extension of EML with higher-order parameterisation at the module level, dependent types, and logically-constrained subtypes.

The following is an EML specification of a generic sorting module called `Sort`. The first axiom in its input interface `PO` requires the function `le` to terminate for any choice of arguments. The first axiom in its output interface `SORT` requires the function `sort` to terminate with a "proper" value (i.e., not an exception) for any argument. In the second axiom of `SORT`, the function `sorted` is a local function whose scope is that axiom. The third axiom in `SORT` assumes that a polymorphic function `count` for counting the number of occurrences of a value in a list has been defined elsewhere.

```
signature PO =
  sig
    eqtype elem
    val le : elem * elem -> bool
    axiom forall(x,y) => le(x,y) terminates
    axiom forall x => le(x,x)
    axiom forall(x,y) => le(x,y) andalso le(y,x) implies x==y
    axiom forall (x,y,z) => le(x,y) andalso le(y,z) implies le(x,z)
  end

signature SORT =
  sig
    structure Elem:PO
    val sort : Elem.elem list -> Elem.elem list
    axiom forall l => (sort l) proper
    axiom let
            fun sorted l =
```

```
            forall (l1,x,l2,y,l3) =>
                l1@[x]@l2@[y]@l3==l implies Elem.le(x,y)
        in
          forall l => sorted(sort l)
        end
    axiom forall l => forall x => count(x,l)==count(x,sort l)
  end

functor Sort(X : PO) : sig include SORT sharing Elem=X end = ?
```

SPECTRUM. SPECTRUM [BFG+93] is a language for developing executable specifications from requirements specifications. It contains explicit support for partial functions, and axioms are not restricted to equational or conditional equational formulas since executability is not the primary aim.

The whole specification style is oriented towards functional programming and the specification of concurrent systems via dataflow nets of (infinite) streams. As a consequence a number of functional language concepts have been integrated into the specification language. The sort system provides parametric polymorphism and sort classes in the style of the functional language Haskell [HPW92]. Furthermore, the language supports the definition of infinite objects by recursion equations, the definition of functions by typed λ-abstractions, and higher-order functions. Functions can be partial, strict, non-strict, continuous. Non-continuous functions are called "mappings" and are only used as auxiliary functions for specification purposes. The constructs for specification in the large and for parameterisation are inspired by ASL.

The semantics of SPECTRUM is loose. Its definition and proof system were influenced by languages and systems of the LCF family [GMW79], cf. [Reg94]. The logic of SPECTRUM is similar to PPλ, the logic of computable functions underlying LCF. In particular, all carrier sets are complete partial orders. The main difference to PPλ is due to the notion of sort class which semantically is treated like the notion of kind in type theory.

The following is a SPECTRUM specification of a sort class `TO` that defines the class of totally ordered sorts. The attributes `strict` and `total` of $\leq$ ensure that $\leq$ is a total function and that applying $\leq$ to a bottom (i.e., undefined) element yields the bottom element (of `Bool`) as result.

```
Torder = {
  class TO;
  .≤. : a :: TO ⇒ a × a → Bool;
  ≤ strict total;
  axioms a :: TO ⇒ ∀x, y, z: a in
  x ≤ x;                              -- reflexivity
  x ≤ y ∧ y ≤ x ⇒ x = y;              -- antisymmetry
  x ≤ y ∧ y ≤ z ⇒ x ≤ z;              -- transitivity
  x ≤ y ∨ y ≤ x;                      -- linearity
  endaxioms;
}
```

The specification `MinLIST` extends `Torder` and a polymorphic specification `LIST1` of lists by a partial function `min` for computing the minimum of a list (cf. `POLY_LIST` in Section 8.3.5; `LIST1` is an extension of `POLY_LIST` by the function $\in$ that checks whether an element is a member of a list). The first axiom ensures that `min` is undefined for the empty list `nil`.

```
MinLIST = {
  enriches Torder + LIST1;
  min : a :: TO ⇒ List a → a;
  min strict;
  axioms a :: TO ⇒ ∀x: a, l : List a in
  ¬(δ(min nil));
  ¬(l = nil) ⇒ min(l) ∈ l ∧ (x ∈ l ⇒ min(l) ≤ x);
  endaxioms;
}
```

CASL. CASL [CoF98,Mos97] is a language for specifying requirements and designs. It was designed by combining the most successful features of existing languages to give a common basis for future joint work on algebraic specifications. CASL is intended to be the central member of a coherent family of languages, with simpler languages (e.g., for use with existing term rewriting tools) obtained by restriction, and more advanced languages (e.g., for specifying reactive systems) obtained by extension.

CASL supports both partial and total functions as well as predicates, with axioms of first-order logic with equality and definedness assertions (Chapter 3). Datatype declarations allow concise specification of sorts together with constructors and (optionally) selectors. Subsorts are provided, including logically-constrained subsorts as in Spectral, and OBJ-style mixfix operator syntax is supported. Structuring operators (translation, reduction, union, loose extension, free extension) and pushout-style generic specifications are available for building large specifications. Architectural specifications describe how to build the specified software from separately-developed generic units with specified interfaces. Specification libraries allow the distributed storage and retrieval of named specifications.

The following CASL specification defines ordered lists over a partially-ordered type of elements as a subsort of ordinary lists.

```
spec PO =
  type Elem
  pred __ ≤ __ : Elem * Elem
  vars x,y,z : Elem
  axioms x≤x ;
         x≤y ∧ y≤x ⇒ x=y ;
         x≤y ∧ y≤z ⇒ x≤z
  end

spec ORDLIST[PO] =
```

```
free type List[Elem] ::= nil | __::__ (hd:?Elem; tl:?List[Elem])
pred ordered : List[Elem]
vars a,b : Elem; l : List[Elem]
axioms ordered(nil) ;
       ordered(a::nil) ;
       a≤b ∧ ordered(b::l) ⇒ ordered(a::(b::l))
type OrdList[Elem] = { l : List[Elem] . ordered(l) }
end
```

The constructor `__::__` is a total function of type `Elem` × `List`[`Elem`] → `List`[`Elem`]. Suppose that a : `Elem` and l : `OrdList`[`Elem`]. Since `OrdList[Elem]` is a subsort of `List[Elem]`, the term a::l is well-formed and is of type `List[Elem]`. We can "cast" this term to type `OrdList[Elem]` by writing a::l `as OrdList[Elem]`. This term is well-formed and has the indicated type but will have a defined value if and only if `ordered`(a::l).

`ORDLIST` is instantiated by supplying an actual parameter that fits the formal parameter `PO`. One way to do this is to supply a specification together with a fitting morphism, for example:

```
ORDLIST[NAT fit Elem |-> Nat, __≤__]
```

which, assuming that `NAT` is a specification of natural numbers with sort `Nat` and predicate ≤, gives ordered lists over the natural numbers with sort names `List[Nat]` and `OrdList[Nat]`. The same effect may be achieved by using a named "view":

```
view PO-NAT : PO to NAT = Elem |-> Nat, __≤__

ORDLIST[view PO-NAT]
```

A view may itself have parameters, which have to be instantiated when it is used.

8.3 Design decisions

8.3.1 Context of the design

Design decisions are influenced by the general context of the design as well as by the taste of the designer and technical feasibility. The context includes the range of uses to which the language will be put, the environment in which the language will be used, and the tools available to support its use. All of these factors dictate to some extent the choices discussed in the following sections.

Range of applications. The intended range of applications is a significant factor in the design of a specification language. A special-purpose language may be able to provide extra support for a given application domain with

additional features to cope with special aspects of the domain. Examples of application domains are: safety-critical systems, where mechanisms to support fault analysis might be required; real-time systems, where notations to indicate timing constraints would be provided; and information systems, e.g., supported via the entity-relationship model.

Level of abstraction. Different specification languages aim to describe problems and/or systems at different levels of abstraction. Possible levels include: that of the problem to be solved (requirements specifications); that of the module structure of the system (design specifications); that of the final solution itself (programs). Some languages focus on a single level of abstraction, while others aim to cope with two or even more of these levels. This influences the form of axioms, the semantic approach, and the type of parameterisation that the language supports.

Programming paradigm. The question of which programming paradigm the specification language aims to support (e.g., imperative, functional, object-oriented, concurrent) exerts a strong influence on all aspects of the design of the language. The state of the art in algebraic specification is such that the most obvious context to which it is appropriate is the development of functional programs with non-trivial data structures, so this is where most experience has been accumulated and where the discussion below will be most directly relevant. There are extensions to handle other paradigms, e.g., Chapter 12 discusses object orientation and Chapter 13 discusses concurrency.

Software development process. The extent to which a specification language will support the software development process is a factor in the design of the language. The rôle of the language in the software development process depends on the particular development model adopted. More than one specification language may be required, with different languages to support different phases of development.

Tools. A pervasive influence on the design of a specification language is the desired degree of tool support. Tools might include parsers, type checkers, theorem provers, browsers, library search tools, rapid prototyping tools, graphical visualization tools, version control mechanisms, analysis tools, etc. If extensive tool support is an important factor in the design, then various design decisions may be influenced by the ease with which the resulting language can be mechanically processed. Some tools or meta-tools (e.g., theorem provers for a logical framework like LF [HHP93] or Isabelle [Pau94]) may already be available; then the language design may be influenced by the constraints built into such tools.

8.3.2 General design principles

Other general design decisions are influenced by the intended mode of use of the specification language and by its underlying semantic concepts.

Readability, writeability, etc. For the intended mode of use one may have different aims such as easy readability, writeability, executability, verifiability, modifiability. These choices often conflict: easy readability and writeability may require high-level, easy-to-use constructs, whereas easy verifiability excludes certain high-level constructs. For example SPECTRUM, EML, and CASL are languages with many high-level features which make them suitable for use in practical applications, but proof is more difficult than, e.g., in LSL or RAP [HG86]. Executability rules out the use of constructs such as unrestricted existential quantifiers; this makes it difficult to express certain abstract properties and therefore inhibits easy writeability.

OBJ3 and CASL support readability by the use of mixfix notation, while OBJ3 and a planned future sublanguage of CASL support executability and verifiability by restricting to conditional equational formulas. ACT ONE was originally designed to be an executable kernel language with a close relation to the underlying mathematical concepts of initial algebra semantics, but after experience with use of the language, a more sophisticated syntax was added to increase readability and useability. LSL sacrifices executability in favour of brevity, clarity, and abstractness. To compensate it provides ways to check and to verify specifications with the help of the LP theorem prover. The absence of a hiding operation makes such support easier to provide.

ASL was designed as a kernel language with high expressive power and orthogonal language constructs. As a consequence it has a well-developed set of proof rules. Because of the expressiveness of observational abstraction, verifiability of specifications containing the observational abstraction operator is still a topic of current research although very considerable progress has been made in the past few years [BH96].

Up to now modifiability has not been taken in account by most languages. Pluss offers syntactic constructs called draft and sketch that are designed to support this feature. Maude supports inheritance and a redefinition mechanism inspired by object-oriented languages. These concepts are semantically well-founded in Maude: inheritance is based on subsorting, and redefinition can be modeled with the structuring operators for renaming and hiding. Through use of a modular natural deduction calculus [Pet94], ASL supports the reuse of proofs when specifications are modified. In formal development using EML, the module system insulates the developer from the need to reprove correctness from scratch when interfaces change in certain ways in the course of development [Var92].

General semantic decisions. The semantic concepts underlying a language are often influenced by its intended use: e.g., executability induces the existence

of initial or free models, while verifiability requires the existence of proof rules that are sound with respect to the semantics.

Typical general semantic decisions are: Should it be possible to write inconsistent specifications, or specifications that can only be realized using non-computable functions? Should every specification have an initial or a free model (e.g., so that term rewriting makes sense)?

- Inconsistent specifications are generally possible with loose semantics, but not with initial semantics where there is always at least one model, which might be trivial. Another way to force consistency is to restrict the form of the axioms to explicit definitions on top of a fixed set of given consistent specifications, as in model-based approaches to specification.
- Models can be forced to be non-computable when full first-order logic is used or when "local" generating or initial constraints are present [Wir90]. ACT ONE protects against these "errors"; most other languages do not. Exceptions are a subset of OBJ3 and other executable specification languages including the executable subsets of SPECTRUM and EML.
- Initial/free models exist for the executable subset of a language where the operational semantics coincides with the mathematical semantics. Typically, this requires conditional equational axioms or more generally Horn or hereditary Harrop formulas as in λProlog [NM88], sufficient completeness with respect to the constructors, and only simple equations (such as associativity, commutativity, and idempotence) between constructors.

The decision depends on the intended use of specifications. If one aims at abstractness for describing requirements of a program or a software system, then the requirement of consistency, existence of initial and free models, and computability of models should be sacrificed. On the other hand, for the design of algorithms and operational descriptions of dynamic behaviour, most of these properties should hold. See [ST96] for an analysis of the relationship between models of specifications and models of programs.

Structure of language definition. A separate question about the semantics concerns the structure of the semantic description of a complex specification language. A convenient option is to define the semantics for a subset of the language or for a more primitive language (the so-called "kernel", which is chosen to be simple, orthogonal, and low-level) and then to translate all syntactic constructs to this kernel (compare the semantics of CIP-L [BBB$^+$85], COLD [FJ92], and RSL [Mil90]). The problem is to choose the right kernel: if it is too small then the translation may be too complex; if it is too large then the semantics of the kernel language may be too complicated. ASL is a kernel language, and EML, Pluss, and SPECTRUM are all built (in principle) on top of this kernel.

8.3.3 Specification in the small

For specification "in the small", the first problem is to decide on a kind of signature, a logical system for writing axioms, and a notion of algebra for defining the meaning of axioms via the definition of satisfaction. Of course, these choices are not independent since signatures provide names for sorts, operators, etc., that appear in axioms and are interpreted by algebras.

Institutions. In most cases, the choices made will satisfy additional compatibility conditions and so will give rise to an institution (see Chapter 4). This has certain advantages for the design of features for specification in the large; see the next subsection. It is possible to postpone most decisions pertaining to specification in the small by deciding on an institution-independent approach, as with Clear and ASL; this avoids premature commitment, and leads to a family of compatible languages. Still, to actually make use of the language it is necessary to choose a particular institution, and then these decisions must be made. If the particular choices that seem most desirable turn out not to give rise to an institution, then either the use of an institution-independent approach or the features chosen for specification in the small will need to be reconsidered. More likely, as in EML and CASL, the "in the large" features depend to a minor extent on the details of the "in the small" features, and then one can employ an appropriately enriched version of institutions to specify the interface between the two language layers. See [ST86] for the case of EML and [Mos98] for the case of CASL.

Signatures and type structure. The main point that needs to be decided for the choice of signatures is that of type structure. Should signatures be one-sorted (i.e., untyped) or many-sorted (i.e., typed) and, if many-sorted, should there be simple sorts, polymorphism, subsorts, higher-order types, or sort classes? The standard choice for specification languages is to have typed signatures. Defining types, which imposes conceptual structure on the problem domain, seems to be a natural part of writing specifications. Although untyped calculi such as the λ-calculus have enormous expressive power, they are difficult to use effectively because the power is in a raw form which makes errors easy to commit and difficult to detect. Type systems are used to tame this raw power by distinguishing well-formed expressions from ill-formed ones; the latter are regarded as meaningless and are therefore rejected. The expressive power of the type system determines how many useful expressions are regarded as well-formed, but there is a trade-off between tractability of typing and expressiveness of types. Simple type systems allow static type checking or even static type inference, and automatic disambiguation of expressions. In complex type systems, type checking amounts to a form of theorem proving which means that error detection cannot be fully automatic. LSL has simple sorts and ACT ONE has simple sorts with structured names (see the example in Section 8.2). In OBJ3, ordered sorts provide increased expressive power but as a result some typechecking must be deferred to run-time

using the mechanism of "retracts"; the same holds for Spectral's system of dependent types, except that typechecking of logically-constrained subsorts requires theorem proving in general. EML inherits its higher-order polymorphic type system from SML, and the same algorithm can be used to infer types automatically. SPECTRUM has higher-order types and sort class polymorphism but no subsorts, and types can be inferred automatically as in Haskell. CASL has ordered sorts including logically-constrained subsorts as in Spectral. A planned future extension of CASL will include polymorphic types and higher-order functions [HKM98].

The difference between these alternatives can be seen by comparing how lists or bags over an ordered domain of values would be specified in each language. The specification of `Sort` in Section 8.2 indicates how this is done in EML; lists over an arbitrary type of values are handled via a (built-in) polymorphic type but the order on the values is dealt with via a module parameter. In SPECTRUM, sort classes would be used to handle both kinds of parameterisation – see the specification `MinLIST` in Section 8.2. Here, polymorphism is a degenerate case of the use of sort classes: `List` α is a shorthand notation for the declaration of a sort constructor `List::(CPO)CPO` where `CPO` is the default class of complete partial orders. Sort constructors can be overloaded, e.g., the additional declaration `List::(EQ)EQ` asserts that if the argument is of sort class `EQ` then the result sort is of sort class `EQ` as well. ACT ONE would use a parameterised type very much like `LIST` in Section 8.2, but with a parameter specification that requires an order on the parameter type. The use of structured sort names in ACT ONE resembles the use of polymorphism in EML, but the resemblance is merely on a syntactic level. The CASL version of this example given in Section 8.2 is similar except for the use of predicates in place of boolean-valued functions and the definition of ordered lists as a logically-constrained subtype of ordinary lists. The treatment in CafeOBJ would be similar to the CASL version except that the subtype of ordered lists would be axiomatised using a sort membership predicate.

LSL would handle this example by using an assumes clause which has to be discharged when the parameter is instantiated, as follows:

```
TO(E): trait
  introduces _≤_: E, E → Bool
  asserts
    ∀ x, y, z: E
      x ≤ x;
      (x ≤ y ∧ y ≤ z) ⇒ x ≤ z;
      (x ≤ y ∧ y ≤ x) == (x = y);
      (x ≤ y ∨ y ≤ x)

BAG_TO (E, Bag): trait
  assumes TO (E)
  includes BAGcount (E, Bag)
```

Let `NAT1` be a trait that extends natural numbers by the usual order relation. Then in

```
NatBAG_TO: trait
  includes
    NAT1, BAG_TO (Nat)
```

the assumption `TO (Nat)` has to be discharged by proving that the ordering on natural numbers is a total order.

Finally, in ASL the way that this example is expressed would depend on the institution in use: in an institution with signatures containing only simple sorts, parameterisation would be used as in ACT ONE; in an institution with ordered sorts, the treatment would be like that in OBJ3; and so on.

Expressiveness of logic. The simplest logic used in algebraic specification is equational logic. This has a simple proof system that is complete and can be easily mechanised, but it only permits simple properties to be conveniently expressed. Therefore additional structuring operators, such as hiding, and additional constraints are necessary to adequately axiomatise some data structures. LSL uses equational logic while OBJ3 and ACT ONE use conditional equational logic. On the other hand a complex logic like higher-order logic is extremely expressive, allowing for example the expression of hiding via higher-order quantification and of generating constraints as ordinary formulas, but no complete proof system can exist and proof search is much less tractable than in simpler logics. EML and SPECTRUM use higher-order logic with equality. A reasonable compromise is to use first-order logic with equality, as in CASL, see Chapter 3. See [Wir90] for results on expressiveness of different first-order specification frameworks. Although there is a trade-off between expressiveness and tractability which is analogous to that in the case of type structure, the choice here also depends on the phase of the development process that is addressed. Ease of expression is crucial for requirements specifications, while equations or conditional equations are appropriate for programs (see, e.g., [GHM88]). Although equational logic is in principle very expressive (see, e.g., [Wir90]), the simplest and most natural way of writing a specification often requires the use of a more complex logic. For example, the function $prime\colon nat \to bool$ for testing primality is easily specified in first-order logic with equality using existential quantification:

$$\forall p : nat.\ (\exists n : nat.\ p > n > 1 \wedge divides(n, p)) \Leftrightarrow prime(p) = false$$

Specifying this function in equational logic or conditional equational logic is possible (with hidden functions), but the specification is much longer and much less straightforward.

Computational phenomena. Another dimension of complexity concerns the way that the logic deals with computational phenomena such as partial functions, non-termination and exceptions. To a large extent this depends on the

context of application, see Section 8.3.1. For example, EML was designed for specifying SML functions, so it needs to cope with exceptions and non-termination. Similarly, one main application area of SPECTRUM is specification of operations on streams (employing non-terminating functions), used to model distributed systems [BDD+92]. Some languages, including ACT ONE, do not attempt to deal with such phenomena and thereby exclude many important examples while avoiding extra complexity. A related choice is that taken by LSL, where all functions are total but the semantics is loose; thus $pop(empty)$ would be a stack, but one about which nothing is known. Another option is to attempt to avoid partial functions by viewing them as total functions on a specified subsort whenever possible; this is the approach taken in OBJ3, Spectral, and CASL. For specification languages that attempt to deal with these phenomena directly, there are many choices. This is a very delicate matter since these choices interact with each other and also with the choice of the notion of algebra. For example, one may choose to deal with non-terminating functions but require all functions to be strict, or one may choose to admit non-strict or even non-monotonic functions; the primitives provided by the logic for making assertions about functions will vary accordingly. One needs a way of expressing definedness in any case, but in the case of non-strict functions one also needs a way of quantifying over both defined and undefined values. The choice of the meaning of equality is a subtle issue, with the choices including *strong equality* $t \stackrel{s}{=} t'$ (the values of t and t' are defined and equal or else are both undefined) and *existential equality* $t \stackrel{e}{=} t'$ (the values of t and t' are defined and equal). See Chapter 3 for the details of many of these options. CASL has a definedness predicate and both strong and existential equalities. If equality is available on function types, there is a further choice between extensional and intensional equality. EML and SPECTRUM both have a definedness predicate and strong extensional equality. EML also has a predicate for distinguishing exceptional values. Another difference between these two languages is quantification: in EML this ranges over SML-expressible values only, while quantification over functions in SPECTRUM can range either over all functions or alternatively over only the continuous functions.

Attributes. It is sometimes convenient to record certain standardized properties of an operation in a specification as attributes attached to its name in the signature rather than by writing an equivalent formula as an axiom. Examples are associativity and commutativity attributes in OBJ3 and CASL and strictness and totality attributes in SPECTRUM. Although both ways of expressing such a property are semantically equivalent, expressing it in the form of an attribute can enable "syntactic" use to be made of this "logical" information. For example, proofs about associative and commutative operations can use AC-unification in place of proof steps appealing to the corresponding axioms. This simplifies proof search, especially since the unrestricted use of permutative properties like commutativity leads to infinite

search paths. Rules of inference for reasoning about strict or total operations are simpler than in the unrestricted case; this again leads to shorter proofs. Such attributes can also be used to restrict the space of possibilities when searching in a library of specifications.

Another kind of attribute that may be recorded with a specification is its intended and/or verified consequences. Examples of such attributes are the property of sufficient completeness in LSL (called converts) and RAP as well as the set of derived or required theorems in LSL (using the implies clause).

Algebras. As mentioned above, the choice of algebras is closely related to the choice of logic. The usual choice is to take the simplest model that correctly captures the aspects of computation that are of interest. For example, since ACT ONE does not attempt to deal with partial functions, etc., ordinary many-sorted algebras suffice for its purposes. If initial semantics is chosen as in ACT ONE, there is another potential point of interaction, since then homomorphisms have to be defined and initial models must exist in order for specifications to be meaningful. For specifying concurrent or reactive systems, one requires more complex models: transition systems, event structures, Petri nets, etc. But then a more general structural view seems to be necessary: see, e.g., [Rut96].

Semantic approach. The last item is the semantic approach (see Chapter 5). The choice of loose or initial semantics depends mainly on the phase of the software development process that specifications are intended to address; see Section 8.3.1. Requirements specifications are naturally loose, while programs are naturally initial. Design specifications might be either loose if they describe only part of the structure of a program, or initial if they give a complete input/output definition. Another point is that initial semantics requires the use of axioms no more complex than conditional equations. Orthogonal to the choice between initial and loose semantics is the choice between a semantics in terms of model classes or a semantics in terms of theories or presentations. Model-class semantics is the more flexible choice but theories and presentations relate more directly to proof. ASL is a requirements specification language (although the extension of ASL described in [SST92] can be used for design specifications as well) and it therefore has a loose semantics, which is expressed in terms of model classes. ACT ONE and ACT TWO are design specification languages; ACT ONE uses initial semantics (thus the restriction to conditional equational axioms) while ACT TWO uses loose semantics for interface specifications and initial semantics for module body specifications. Both are given in terms of both presentations and models. LSL defines concepts for use in Larch interface language annotations in finished programs or program modules; both languages have loose semantics, expressed in terms of theories. Ordinary structured specifications in CASL are for specifying requirements while architectural specifications are for specifying designs, and the semantics of both are loose and are expressed in terms of model classes.

OBJ3, EML, and SPECTRUM all allow the definition of executable programs as well as the specification of their interfaces. In each case the semantics of interface specifications is loose, while for programs the computation mechanism can be seen as choosing a particular algebra in the class of models described by the program.

8.3.4 Specification in the large

It is generally agreed that specification languages need to supply mechanisms for building large specifications in a modular way from small units.

In the small = in the large? The first question to ask is whether these mechanisms need to be separated from the mechanisms for specification in the small. All algebraic specification languages impose such a separation, and this is natural from the point of view of modular programming where the mechanisms provided for building and composing modules are different from the mechanisms provided for coding algorithms and data structures. However, if the type system used in signatures is sophisticated enough, these levels can be merged. An example is ECC [Luo90], where, e.g., dependent sums are used for tuples of all kinds: tuples of values, tuples of module components, pairs of a signature and a list of axioms, etc. Here there is not even a distinction between signatures and axioms because propositions are regarded as types having proofs as values. In spite of the elegance of a compact framework, it can be argued that the separation between levels provides conceptual clarity and contributes to ease of understanding.

Institution independence. If an institution-independent approach is taken, the features for specification in the large must be defined completely independently of the features for specification in the small, since these will vary from one institution to another. This is related to the principle of orthogonality in programming language design, and seems to be a useful way of decomposing the design of a specification language. Variants of most standard specification-building operations, such as those in Clear and ASL, are available for use in an arbitrary institution [ST88a], so this approach does not impose strong restrictions on the structuring features of the language. These operations are relatively well-understood, and come equipped with proof rules for reasoning about specifications built using them. Although OBJ3 and EML are not institution-independent, much of their underlying foundation has been developed in this framework via Clear and ASL respectively. For example, the theory that underlies EML's methodology for formal development is entirely institution-independent. CASL is not institution-independent but its features for specification in the large are defined for an arbitrary institution enriched with elements for handling compound identifiers and presentations of signature morphisms [Mos98].

Choice of specification-building operations. Some of the tradeoffs involved in the choice of particular specification-building operations are similar to those in the choice of logic. Decreased expressive power generally leads to increased tractability. For example, proving properties of specifications written in a language with only simple specification-building operations such as union and/or enrich, and rename (e.g., OBJ3, LSL or ACT ONE) is easier than when operations such as observational abstraction are used. In the former case it is possible to translate specifications into "flat" presentations, reducing the problem to ordinary theorem proving; in the latter case separate mechanisms are required for performing proofs in structured specifications. In other words, the semantics of a language with simple specification-building operations can be given in terms of presentations, while the semantics of a language like ASL requires the use of model classes. On the other hand, the structure of a specification conveys information about the conceptual structure of the problem domain, and it seems useful to take advantage of this structure in understanding and analyzing specifications rather than destroying it via flattening. However, a language with simple operations typically requires the use of a complex notion of implementation (Chapter 7) which complicates the problem of proving correctness of implementation steps. The use of observational abstraction makes it possible to identify the class of models of a specification with its class of realizations and to use model class inclusion for refinement. In fact, this does not truly simplify the task of proving correctness of implementations; it just shifts the problem to that of proving properties of specifications. Another possibility is to shift the complexity to the logic used for writing axioms [Wir93], which makes ordinary theorem proving more difficult.

Enrichment. All specification languages contain some form of enrichment, extension or importation construct for building a new specification SP' by adding sorts, operations, and/or axioms to what is already contained in an existing specification SP. But this denotes subtly different things in different languages. In the initial algebra approach embodied in ACT ONE (compare data enrichment in Clear and including/using in OBJ3) it denotes the free construction. In the loose approach used in ASL, LSL, SPECTRUM, EML, and CASL (compare "ordinary" enrichment in Clear), it denotes an extension where taking the reduct of a model of SP' to the signature of SP gives a model of SP. Loose and free extension do not protect the meaning of the original specification SP. The free construction may add new values to the existing sorts of SP (if it is not "sufficiently complete"), or cause existing values in such sorts to become equated (if it is not "hierarchy consistent"). In the loose approach neither of these can arise, but it may happen that some models of SP cannot be extended to models of SP'. In an attempt to avoid such problems, OBJ3 introduces two special ways to import modules: extending is supposed to guarantee that new values are not added to existing sorts, and protecting is also supposed to ensure that existing values are not equated,

i.e., that the extension is "persistent". However, the implementation of OBJ3 does not check that these properties actually hold. Similarly, LSL traits may contain assertions that given operations are defined in a sufficiently complete way, and these can be checked using LP. In general, proving persistency is difficult and needs model-theoretic methods unless the new axioms in SP' take the form of explicit definitions. The consequent difficulty in providing tool support for persistency checking might constitute a reason for excluding such assertions from a specification language.

Sharing. When a specification *SP* is built by combining specifications *SP1* and *SP2*, there is a potential problem if both *SP1* and *SP2* contain a subspecification SP'. It is not desirable to obtain two "copies" of the common sorts and operations; this is not only untidy, but also makes it difficult to express certain properties that involve operators from both *SP1* and *SP2*. For example, if SP' is a specification of natural numbers, *SP1* contains an operation $f1 : s1 \to nat$, *SP2* contains an operation $f2 : nat \to s2$, and *SP* contains two copies of *nat*, then an axiom containing a subexpression of the form $f2(f1(\ldots))$ will not typecheck. There are three basic approaches to avoiding this situation. The first is to take the *disjoint union* of *SP1* and *SP2*, but with some way of explicitly equating sorts and/or operations that should be shared. This is the approach taken in EML module interface specifications, following SML, using so-called "sharing constraints". The second approach is to take the *set-theoretic union*, identifying a symbol in *SP1* with any symbol in *SP2* having the same name (and same type for operations, if overloading is permitted). This may lead to unintended sharing, so mechanisms may be provided for detecting when the sharing is accidental. For example, "origin consistency" [Jon89] would reject an attempt to combine *SP1* with *SP2* when this would identify two symbols that were originally declared in different specifications. For requirements specification this is too restrictive since one may want, e.g., to build a specification for an equivalence relation from specifications of reflexivity, symmetry, and transitivity where the same relation symbol occurs in all three subspecifications. Set-theoretic union is used in ASL, Spectrum, LSL, and Casl. Finally, one can take an *amalgamated sum* of *SP1* and *SP2* with respect to a list of shared subspecifications. This is the approach taken in Clear, OBJ3, and ACT ONE, where the list of shared subspecifications is determined implicitly (compare module body specifications in EML); an alternative is to supply this list explicitly.

Hiding. A specification language requires some means of hiding auxiliary parts of signatures in order to provide adequate expressive power, at least when a logic weaker than second-order logic is used for writing axioms. Hiding can be provided as a specification-building operation, as in the case of derive in Clear and ASL, local in EML and Casl, export/hide in Spectrum, and reveal/hide in Casl. Alternatively (or additionally), hiding can be achieved via an interface attached to a specification, as with hiding via a signature in

EML or "unit reductions" in CASL architectural specifications. ACT ONE has no hiding mechanism but this facility is provided by the "export part" of an ACT TWO module. Similarly, LSL contains no hiding mechanism because an LSL trait is not intended to be implemented itself but only to provide concepts for use in Larch interface language specifications. Hiding complicates modular proofs, and completeness of a modular proof system for a language with hiding requires an interpolation property for the logic in use [BHK90, Cen94]. The use of an interface for hiding takes explicit account of the fact that reasoning about a specification requires access to hidden components.

Constraints. Constraints bridge the realms of specification in the small and specification in the large. Constraints are sometimes viewed as (complicated) axioms, as in SPECTRUM and LSL, and sometimes as specification-building operations, as in ASL. They are sometimes viewed in both ways, i.e., as specification-building operations that correspond semantically to axioms, as in Clear and CASL, where the axioms obtained may not be expressible in the logic used for writing ordinary axioms. In EML, constraints are part of the type system (via `datatype` declarations). In Pluss and OBJ3, the type of a specification determines how a constraint is to be interpreted; for example, in a Pluss `sketch` a constrained sort is required to contain *at least* the values generated by the operations, while in a `spec` it is required to contain *exactly* these values.

Multiple institutions. It is often convenient to use specialized logical tools to reason about particular aspects of a system. For example, automata theory is useful for reasoning about control and communication subsystems, and numerical analysis is useful for reasoning about scientific calculations; if both things co-exist within a single system then there may be a need for integrating properties of one with properties of the other. Related to this is the problem of describing a multi-paradigm system built from heterogeneous components, such as mixed hardware/software systems. It may also be appropriate to use different institutions at different stages of development of a system to express properties of a system viewed at different levels of abstraction. The simultaneous use of multiple institutions requires some way of relating the institutions to each other, e.g., via the concept of "institution semi-morphism" that maps the models of one institution (the more "concrete" one) to those of another (the more "abstract" one) [ST], see also [MM95]. None of the languages mentioned earlier have mechanisms for the use of multiple institutions, although [ST88b] describes an extension to ASL that permits specifications in one institution to be translated to another institution via an institution semi-morphism; cf. [Tar98] and [ST].

8.3.5 Parameterisation

The intention of parameterisation in a specification language is to abstract away from a part of the specification in order to be able to instantiate it with

different data types or specifications. This allows specifications to be defined in a generic fashion so that they may be applied in a variety of contexts that share some common characteristics. See Chapter 6.

Parameterisation is normally regarded as an important specification-structuring mechanism and therefore it really belongs in the previous section. It is treated separately here only in order to keep it from dominating the other topics in that section.

Implicit versus explicit parameterisation. Syntactically, one can distinguish between explicit parameterisation where the parameter part is fixed by means of some particular syntactic notation, and implicit parameterisation where only an instantiation mechanism is provided without any particular notation for abstraction.

Implicit parameterisation is often used in languages for requirements specification in order to re-use previously defined specifications. Implicit parameterisation can be seen as just another use of the renaming mechanism (cf. Section 8.3.4) in languages without explicit parameterisation, such as early versions of LSL.

In explicit parameterisation there is a syntactic construct for abstraction consisting of a formal parameter and a body. The formal parameter is usually a specification that is constrained by a signature and a set of axioms. In (the new version of) LSL it is a list of sort names and operation names. The syntax is similar in all specification languages. Compare:

```
proc LIST(X : ELEM) =
              enrich X by data sorts List ... enden     in Clear
LIST = param X = ELEM; body {enriches X; ... }          in SPECTRUM
obj LIST[X :: ELEM] is sorts List ... endo              in OBJ3
type LIST[D] is sorts List(D) ... endtype               in ACT ONE
LIST = λX:ELEM. enrich X by ...                         in ASL
spec LIST[ELEM] = free type List[Elem] ::= ... end      in CASL
```

where `ELEM` is a specification containing just the single sort `Elem`. Explicit parameterisation mechanisms are used for the design of generic components (such as in ACT TWO, COLD, and CASL); often it is influenced by the parameterisation concepts of the underlying programming language (as, e.g., in EML).

Pushout-style versus λ-calculus style parameterisation. In pushout-style parameterisation, which is used in Clear, OBJ3, ACT ONE, and CASL, a parameterised specification consists of a "requirement" specification R together with a "body" specification B that extends R. Instantiation requires an actual parameter specification A that "fits" R, witnessed by a specification morphism from R to A, and the result is obtained by a pushout construction.

On the other hand, in view of the fact that a parameterised specification is a (specification-valued) function, a version of the typed λ-calculus may

be used, with λ-abstraction for defining parameterised specifications and instantiation corresponding to β-reduction. This is the approach that is used in ASL, COLD, Spectral, and Spectrum. Then many features of the classical λ-calculus carry over to parameterised specifications; in particular, higher-order parameterisation is possible (see [Wir86,SST92,Cen94]) and (by using uncurrying) specifications with several parameters can be defined. The difference with respect to the classical λ-calculus concerns mainly the type system used. The class of admissible parameters may be restricted to specifications that fit a requirement specification (as in ASL) or may be dependent on former parameters (as in Spectral).

For a large subclass of the first-order case, both styles of parameterisation coincide. In particular, if the body of the parameterised specification is an extension of the requirement specification, the combination of renaming with β-reduction yields the same textual result as the pushout construction. But note that in general specification expressions may be more complicated than simple extension. In these cases the λ-calculus approach is more general than the pushout approach.

Parameterised specifications versus specification of parameterised program modules. Parameterisation can be used both at the level of program modules as in Ada and SML and at the level of specifications, and specifications can describe both ordinary (non-parameterised) program modules and parameterised program modules. Thus it is possible to give a parameterised specification of a non-parameterised program module, or a (non-parameterised) specification of a parameterised program module [SST92]. Most specification languages provide only one of these possibilities (Clear, OBJ3, EML, ASL, LSL, Spectrum) but some provide both. In those languages that provide both, the two concepts may be identified (ACT ONE, Pluss) or viewed as distinct (Spectral, ASL+, Casl). Higher-order parameterisation of either or both kinds may be available (ASL, ASL+, Spectral, Spectrum) or unavailable (Clear, OBJ3, EML, LSL, ACT ONE, Pluss, Casl).

Parameterised specifications provide a flexible mechanism for structuring specifications. These are most appropriate at the level of requirements specifications. Specifications of parameterised program modules define the module's input and output interfaces. These are most appropriate for use in writing design specifications [BST99]. Moreover, this concept ensures a number of compositionality and consistency properties. Languages that identify both concepts support the view (either implicitly or explicitly) that the structure of a requirements specification should match the structure of the eventual implementation. Forcing such a match seems to be inappropriate since structure at the two levels may arise for quite different reasons, see [FJ90].

Polymorphism versus parameterisation. An important design decision is whether both polymorphism and parameterisation should be supported by a specification language. Polymorphism (and even more, sort classes) models

the concepts of generic functions and generic sorts "in the small", whereas parameterisation of specifications models the same concepts "in the large".

A typical example is polymorphic lists. Consider the specification POLY_LIST in SPECTRUM:

```
POLY_LIST = {
  sort List a;
  nil : → List a;
  cons : a × List a → List a;
  cons strict total;
  List a generated by nil, cons;
}

POLY_LIST_NB = POLY_LIST + NAT + BOOL
```

and the parameterised specification LIST in Clear:

```
proc LIST (X : ELEM) =
  enrich X by
    data sorts List
         opns  nil: List
               cons: Elem, List -> List
  enden

LIST_NAT = LIST(NAT[Elem is Nat])
LIST_BOOL = LIST(NAT[Elem is Bool])
```

The main difference is in the way that instantiations are made. Using polymorphism, several instantiations can be made within the same specification, whereas using parameterisation the same thing requires the construction of separate (and sometimes many separate) instantiations of the parameterised specification. Polymorphic languages like SPECTRUM and EML offer both concepts, where parameterisation is mainly included for interface descriptions in design specifications.

The following example shows a combination of polymorphism with parameterisation in SPECTRUM where POLY_LIST is used as a parameter:

```
MAP =
  param
    X = POLY_LIST;
  body {
    enriches X;
    map : (a → b) → List a → List b;
    axioms ... ;
    endaxioms;
  }
```

Subsequently, MAP can be instantiated, e.g., with polymorphic specifications of stacks and queues.

Higher-order functions versus parameterisation. Higher-order functions in specifications can sometimes be expressed using (first-order) parameterised specifications. This is used in languages such as OBJ3 that rely on first-order functions. The restriction to first-order functions is regarded by some as an advantage rather than a limitation [Gog90], but this is a controversial issue. For example, the higher-order function `map` above can be expressed in OBJ3 as follows (this is for the special case where `a` = `b`; the case where `a` ≠ `b` is similar but notationally more cumbersome, see [GM]):

```
obj UNARY is
  sort Elem .
  op f : Elem -> Elem .
endo

obj MAP [X :: UNARY] is
  protecting LIST[X] .
  op map: List -> List .
  var Y : Elem . var L : List .
  eq map(nil) = nil .
  eq map(cons(Y, L)) = cons(f(Y), map(L)) .
endo
```

The main difference is once more in the way that parameter passing is done. With higher-order functions, several higher-order function calls can be made within the same specification whereas the parameterised approach requires the construction of separate instantiations of the parameterised specification. With higher-order functions, the use of higher-order logic is natural (which makes reasoning more difficult) although it is not necessarily forced, while in the parameterised approach first-order logic suffices.

8.3.6 Syntax

Decisions involved in designing the syntax of a specification language are similar to those in programming language design.

Expression syntax. Some specification languages (e.g., EML) support infix operators while others (e.g., OBJ3, Pluss, and CASL) support a very flexible mixfix notation. The trade-off is between a simple syntax that is easy to parse but difficult to read, and a flexible syntax requiring a powerful and extensible parser. In the context of algebraic specifications the use of at least infix operators seems unavoidable, and greater flexibility seems worth the effort: most specifications are written in order to be read by humans, in contrast to most programs. A proviso is that unrestricted support for mixfix syntax requires the use of a parser for a general context-free language, introducing performance penalties that would probably be regarded as prohibitive in a compiler for a programming language. Haskell's "offside rule", where indentation is used in place of parentheses, would probably be a worthwhile additional feature of specification language syntax.

Name structure. A basic design decision is the name structure to be used in large specifications. Most specification languages have a flat name structure (e.g., OBJ3, SPECTRUM, ASL, LSL, CASL) whereas EML supports SML's "dot" notation, where reference to a name n in a specification SP requires the qualified name $SP.n$. When specifications can be nested as in EML, this leads to names such as $A.B.C.n$. A flat name structure, in which every name is global, is impractical for large specifications: it requires the user to keep track of all names in order to avoid name clashes. A possible solution is via the use of origin consistency to detect accidental name clashes, see Section 8.3.4; this guarantees that names can be disambiguated even when the same name is used for different things in different specifications. A related concept is that of overloaded operation names, where types are used instead of origins for disambiguation.

8.3.7 Software development process

The integration of a specification language in a software process model and the choice of the intended programming paradigm are important decisions for the use of the language in software development.

Software process model. In classical software process models such as the waterfall model or its refinements, as well as in software developments methods based on structured analysis or object-orientation, formal specification does not play an important rôle (see, e.g., [Som92]). This situation could change, however, if formal methods can be integrated with pragmatic techniques and if practically usable tool support for formal specification becomes widely available. Formal specifications are useful as a meta-tool for the formal foundation of pragmatic development models. For example, in [Hußß94b, Hußß94a] the pragmatic notations (including diagrams) of SSADM have been axiomatised in SPECTRUM which has led to several proposals for improvements of SSADM.

There are several software development methods based on formal approaches including the transformational approach of the CIP [BBD+81] and PROSPECTRA [HK93] projects (see Chapter 14), the B-method [ALN+91], and the methods used in the projects RAISE [NHWG89], KORSO [PW95], and PROCOS [BLH92]. One of the principal design decisions is whether there should be just one language supporting the whole software development process. In this case the language has to incorporate features for describing requirements, designs, and programs. This leads to wide-spectrum languages such as CIP-L, COLD, RSL, and EML with complex semantics and proof rules. An alternative is to decouple the specification language from the programming language, as in LSL (via the use of a separate interface language) and CASL. On the other hand, if several languages and notations are used, as in the case of most pragmatic approaches, then one has to fix the semantic relationships between these notations, derive correctness criteria and provide

translations between the different languages. For example, one might want to use SPECTRUM for the requirements specification of a system and then ACT ONE for the design specification and a functional language for coding [WDC+94]. An important issue for the practical acceptance of formal techniques in the design and requirements phase is the graphical representation of formal descriptions (see, e.g., [CEW93]) and the integration of formal texts with usual semi-formal descriptions and diagrammatic notations (see [Wol94]).

For the support of the development and maintenance of specifications, some languages include syntactic features for describing the evolution of a specification and the modification of existing specifications for new purposes. For example, Pluss offers different syntactic constructs for specifications that are still under development and for complete specifications (draft/sketch versus spec). Renaming, hiding, and parameterisation provide means for controlled modification. Moreover some languages support refinement as an expressible relation between specifications (e.g., COLD) whereas others consider it as an external meta-relation (e.g., Clear, ASL).

Programming paradigm. "Pure" algebraic specification languages support the algebraic programming style. Roughly, an algebraic program can be characterized as an executable specification that has first-order positive conditional equations as axioms. In this sense all languages that admit only such axioms (e.g., OBJ3, RAP, ACT, ASF [BHK89], and appropriate executable subsets of other languages) can be considered as algebraic programming languages.

In order to support classical programming paradigms too, one has to extend a pure specification language with appropriate language constructs or to design a specification language that directly integrates the algebraic specification style with programming concepts. This is true even in the case of functional programs, which are closely related to the algebraic specification style. For example, higher-order function types are introduced in EML, Spectral, and SPECTRUM. Object-oriented specification languages like Maude and OS [Bre91] employ subsorting and structuring facilities for (class) inheritance. COLD introduces operators of dynamic logic for writing pre- and postconditions of imperative programs, whereas SMoLCS [AR87] and Maude use labeled transition systems (SMoLCS) and term rewriting (Maude) for specifying sequential and concurrent state transitions. Moreover, languages that support imperative, object-oriented, or concurrent programming include linguistic features for sequential or concurrent control flow.

Some specification languages such as EML and Larch relate directly with a specific programming language rather than merely with a programming paradigm. In this case the semantics of both languages have to be integrated. This may pose difficult semantical problems [KS98]. For EML, a verification semantics had to be developed that extends the structural operational semantics of SML by the semantic foundations of ASL-like specifications.

Support for refinement. When constructing a software system it is not sufficient to construct just one specification: one has to develop different specifications ranging from abstract requirement specifications and design specifications to executable specifications or programs. A basic design decision is whether the notion of refinement should be an expressible relation or an external meta-relation. If refinement is expressible, then the difference between specification-building operations and the refinement relation is a matter of one's point of view.

In pure specification languages, such as ASL, Clear, or ACT ONE, a specification is intended to describe only one level of abstraction of a software system at a time. Notions of refinement and implementation have been studied for these languages (see, e.g., [SW82,SW83,Far92,Cen94]), but they are used on the meta-level in the specification development methodology. For example, the development graph of SPECTRUM (see [PW95]) relies on the notion of refinement as model class inclusion, where a specification SP' is a refinement of a specification SP if any model of SP' is also a model of SP.

Some languages, including COLD, ACT TWO, OBJ3, EML, ASL+, Spectral, and CASL, offer features for representing two different levels of abstraction. COLD and ACT TWO introduce notions of "component" (in COLD) and "module" (in ACT TWO) that relate internal and external descriptions of a software system. In COLD one writes `COMP` x`:`K `:=` L for describing a so-called "design component" with name x, interface specification K, and specification L acting as implementation of K; the requirement is that L is a refinement of K. This is similar in EML and Spectral, except that parameterised components are supported (including higher-order parameterisation in the case of Spectral) and L is only required to refine K "up to behavioural equivalence". In ACT TWO a module consists of four parts: export, import, parameter, and body. As in COLD, the body is required to be a refinement of the export part. OBJ3 has the concept of "view" for defining fitting morphisms used for parameter passing. This can also be used for representing a refinement between an OBJ3 theory and an OBJ3 object.

8.4 Future trends

Specification languages that have been under development during the last few years show a number of tendencies in the development of algebraic specification languages. There is a tendency to aim at covering the whole software development process starting from requirements up through executable specifications. Thus they tend to include elements of both loose specifications (for requirements) and initial algebra specifications (in an executable sublanguage). All of them support partial functions in some form, and there is a tendency to favour simple specification-building operations. All of them contain concepts for explicit parameterisation and there is a growing tendency to provide support for specification of parameterised programs. There is a trend to

include features oriented towards particular programming paradigms, mainly for functional and object-oriented programming.

Though currently studied in the literature, certain aspects of programming are not addressed by present algebraic specification languages:

1. The integration of the diagrammatic features of current software engineering notations such as UML [UML97].
2. The integration of application-dependent features such as description techniques for information systems [Tse91,Hußß94b,Hußß94a] or for real-time systems [ZHR92].
3. The combination of different logics in a single specification language for the description of heterogeneous systems [AC93,CM93,Tar98].
4. The specification of dynamic systems and the description of dynamic properties of systems such as safety and liveness; see Chapter 13.

In each case, more work is needed both on the foundations and on experiments with applications.

Inventing languages is a seductive activity. The history of programming languages is littered with languages that were never used by anybody other than their inventors, and the shorter history of specification languages threatens to show similar tendencies. In particular, the strong focus of attention in the past has been on the foundations of specification languages and there has therefore been comparatively little work on putting them into practice. This is an issue that needs to be addressed in the future for algebraic specification in general. One effort in this direction is the Common Framework Initiative (CoFI) which developed CASL as the basis of an attempt to create a focal point for future joint work on algebraic specifications and a platform for exploitation of past and present work on methodology, support tools, etc. [Mos97]. For specification languages in particular, acceptance by software engineers requires much more work on finding an appropriate compromise between purity and practicality, between expressiveness and simplicity of learning and use, and on the development of tools that provide truly useful support to the users of such languages. Today formal specification is practicable for medium-sized applications (see, e.g., [GM94]) but it not suitable for writing, e.g., a first sketch of the structure of a system or for describing standardized architectures as used in large application systems. One possible direction for future development to address this point is graphical representation of static and dynamic aspects of the system under development.

Acknowledgements. Thanks to Didier Bert, Răzvan Diaconescu, Chris George, Bernd Krieg-Brückner, and Horst Reichel for helpful comments.

Bibliography

[ABB+86] E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, J. Storbank Pedersen, G. Reggio, and E. Zucca. The draft formal definition of Ada. Deliverable 7 of the CEC-MAP project, 1986.

[AC93] E. Astesiano and M. Cerioli. Relationships between logical frameworks. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification, Proc. Workshop on Specification of Abstract Data Types ADT'91*, volume 655 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 1993.

[ALN+91] J.-R. Abrial, M. Lee, D. Neilson, P. Scharbach, and I. Sørenson. The B-method. In S. Prehn and W.J. Toetenel, editors, *VDM, Formal Software Development Methods, Proc. 4th Intl. Symposium of VDM Europe; Vol. 2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 1991.

[AR87] E. Astesiano and G. Reggio. SMoLCS-driven concurrent calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT'87, Vol. 1*, volume 249 of *Lecture Notes in Computer Science*. Springer, 1987.

[Asp97] D. Aspinall. *Type Systems for Modular Programs and Specifications*. PhD thesis, University of Edinburgh, 1997.

[BBB+85] F. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, H. Wössner, and M. Wirsing. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer, 1985.

[BBD+81] F. Bauer, M. Broy, W. Dosch, R. Gnatz, B. Krieg-Brückner, A. Laut, M. Luckmann, T. Matzner, B. Möller, H. Partsch, P. Pepper, K. Samelson, R. Steinbrüggen, H. Wössner, and M. Wirsing. Programming in a wide spectrum language: a collection of examples. *Science of Computer Programming*, 1:73–114, 1981.

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The design of distributed systems: an introduction to FOCUS. Report TUM–I9203, Institut für Informatik, Technische Universität München, 1992.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, and K. Stølen (The Munich SPECTRUM Group). The requirement and design specification language SPECTRUM: An informal introduction, Version 1.0, Part I. Technical Report TUM–19311, TUM–19312, Institut für Informatik, Technische Universität München, 1993.

[BG77] R. Burstall and J. Goguen. Putting theories together to make specifications. In *Proc. 5th Intl. Joint Conference on Artificial Intelligence, Cambridge, Mass. (USA)*, pages 1045–1058, 1977.

[BG80] R.M. Burstall and J.A. Goguen. The semantics of CLEAR, a specification language. In D. Björner, editor, *Proc. Copenhagen Winter School*

on Abstract Software Specification, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer, 1980.

[BGM89] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable? An experiment with the PLUSS specification language. *Science of Computer Programming*, 12(1):1–38, 1989.

[BH96] Michel Bidoit and Rolf Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165(1):3–55, 1996.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series. Addison-Wesley, 1989.

[BHK90] J.A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372, 1990.

[Bid89] M. Bidoit. *Pluss, un langage pour le développement de spécifications algébriques modulaires.* Thèse d'Etat, Université de Paris-Sud, 1989.

[BLH92] D. Bjørner, H. Langmaack, and C.A.R. Hoare. Provably correct systems. PROCOS I final deliverable, 1992.

[Bre91] Ruth Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *Lecture Notes in Computer Science*. Springer, 1991.

[BST99] M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. In *Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology (AMAST'98)*, Lecture Notes in Computer Science. Springer, 1999.

[Cen94] María Victoria Cengarle. *Formal Specifications with Higher-Order Parameterization.* PhD thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 1994.

[CEW93] I. Claßen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development.* AMAST Series in Computing, Vol. 1. World Scientific, 1993.

[Cla89] I. Claßen. Revised ACT ONE: categorical constructions for an algebraic specification language. In *Proc. Workshop on Categorical Methods in Computer Science with Aspects from Topology*, volume 393 of *Lecture Notes in Computer Science*, pages 124–141. Springer, 1989.

[CM93] M. Cerioli and J. Meseguer. May I borrow your logic? In *Proc. 18th Intl. Symp. on Mathematical Foundations of Computer Science MFCS'93*, volume 711 of *Lecture Notes in Computer Science*, pages 342–351. Springer, 1993.

[CoF98] The CoFI Task Group on Language Design. CASL – the common algebraic specification language – summary (version 1.0). Available at `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`, 1998.

[DF98] R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* World Scientific, 1998.

[Far92] Jordi Farrés-Casals. *Verification in ASL and Related Specification Languages.* PhD thesis, University of Edinburgh, 1992. Report CST-92-92.

[Fey88] W. Fey. *Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language*. PhD thesis, Technische Universität Berlin, 1988. Report 88/26.

[FJ90] J.S. Fitzgerald and C.B. Jones. *Modularizing the formal description of a database system*, volume 428 of *Lecture Notes in Computer Science*. Springer, 1990.

[FJ92] L.M.G. Feijs and H.B.M. Jonkers. *Formal specification and design*. Cambridge University Press, 1992.

[GH86] J. Guttag and J. Horning. Report on the Larch shared language. *Science of Computer Programming*, 6:103–134, 1986.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[GHM88] A. Geser, H. Hußmann, and A. Mück. A compiler for a class of conditional term rewriting systems. In *Proc. Intl. Conf. on Conditional Term Rewriting Systems (CTRS'87)*, volume 308 of *Lecture Notes in Computer Science*, pages 84–90. Springer, 1988.

[GM] J. Goguen and G. Malcolm. More higher order programming in OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. To appear.

[GM94] G. Guiho and F. Meija. Operational safety critical software methods in railways. In *Information Processing '94, Vol. 3*, pages 262–269. North-Holland, 1994.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[Gog90] J. Goguen. Higher-order functions considered unnecessary for higher-order programming. In D. Turner, editor, *Research Topics in Functional Programming*, pages 309–352. Addison-Wesley, 1990.

[GWM+92] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ3. Technical Report SRI-CSL-92-03, SRI International, 1992.

[Hen97] Rolf Hennicker. Structured specifications with behavioural operators: Semantics, proof methods and applications. Habilitation Thesis, Ludwig-Maximilians-Universität München, 1997.

[HG86] H. Hußmann and A. Geser. Experiences with the RAP system – a specification interpreter combining term rewriting and resolution. In *European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 339–335. Springer, 1986.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[HK93] Berthold Hoffmann and Bernd Krieg-Brückner, editors. *Program Development by Specification and Transformation*, volume 680 of *Lecture Notes in Computer Science*. Springer, 1993.

[HKM98] A.E. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Extending CASL with higher-order functions – design proposal. CoFI note: L-8, available at `http://www.brics.dk/Projects/CoFI/Notes/L-8/index.html`, 1998.

[HPW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the programming language Haskell, a non-strict, purely functional language (version 1.2)*. SIGPLAN Notices 27(5). 1992.

[Huß94a] H. Hußmann. Formal foundations for pragmatic software engineering methods. In [Wol94], pages 27–34. 1994.

[Huß94b] H. Hußmann. *Formal Foundations for SSADM*. Habilitation thesis, Technische Universität München, 1994.

[HWB97] Rolf Hennicker, Martin Wirsing, and Michel Bidoit. Proof systems for structured specifications with observability operators. *Theoretical Computer Science*, 173:393–443, 1997.

[Jon89] H.B.M. Jonkers. Description algebra. In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, volume 394 of *Lecture Notes in Computer Science*, pages 283–328. Springer, 1989.

[KS91] B. Krieg-Brückner and D. Sannella. Structuring specifications in-the-large and in-the-small: higher-order functions, dependent types and inheritance in SPECTRAL. In *Proc. Colloq. on Combining Paradigms for Software Development, Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, volume 494 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 1991.

[KS98] S. Kahrs and D. Sannella. Reflections on the design of a specification language. In *Proc. Intl. Colloq. on Fundamental Approaches to Software Engineering. European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, volume 1382 of *Lecture Notes in Computer Science*, pages 154–170. Springer, 1998.

[KST94] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Report ECS-LFCS-94-300, University of Edinburgh, 1994.

[KST97] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[Luo90] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. Report CST-65-90.

[Mes93] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[Mil90] R. Milne. The semantic foundations of the RAISE specification language. Technical Report REM/11, RAISE project, STC Technology Ltd, 1990.

[MM95] J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop*, volume 906 of *Lecture Notes in Computer Science*, pages 48–80. Springer, 1995.

[Mos97] Peter D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 1997.

[Mos98] T. Mossakowski. Institution-independent semantics for CASL-in-the-large. Technical Report S-8, The Common Framework Initiative, 1998. Available from `http://www.brics.dk/Projects/CoFI/Notes/S-8`.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NHWG89] M. Nielsen, K. Havelund, K.R. Wagner, and C. George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1:85–114, 1989.

[NM88] G. Nadathur and D. Miller. An overview of λprolog. In *Proc. 5th Intl. Logic Programming Conference, Seattle*, pages 810–827. MIT Press, 1988.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[Pau96] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[Pet94] H. Peterreins. A natural deduction calculus for structured specifications. Report 9410, LMU München, 1994.

[PW95] P. Pepper and M. Wirsing. A method for the development of correct software. In M. Broy and S. Jähnichen, editors, *KORSO: Correct Software by Formal Methods*, Lecture Notes in Computer Science. Springer, 1995.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994.

[Rut96] J.J.M.M. Rutten. Universal coalgebra: A theory of systems. Report CS-R9652, CWI, SMC Amsterdam, 1996.

[San91] D. Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement, Hursley Park*, Springer Workshops in Computing, pages 99–130, 1991.

[SB83] Donald Sannella and R.M. Burstall. Structured theories in LCF. In G. Ausiello and M. Protasi, editors, *Proc. 8th Colloquium on Trees in Algebra and Programming (CAAP)*, volume 159 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 1983.

[Som92] I. Sommerville. *Software Engineering*. Addision Wesley, 4th edition, 1992.

[SST92] Donald Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29(8):689–736, 1992.

[ST] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*. Cambridge University Press. To appear.

[ST86] Donald Sannella and Andrzej Tarlecki. Extended ML: An institution-independent framework for formal program development. In *Proc. Workshop on Category Theory and Computer Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 364–389. Springer, 1986.

[ST88a] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[ST88b] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25(3):233–281, 1988.

[ST91] D. Sannella and A. Tarlecki. Extended ML: past, present and future. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification, Proc. 7th Workshop on Specification of Abstract Data Types*, volume 534 of *Lecture Notes in Computer Science*, pages 297–322. Springer, 1991.

[ST96] Donald Sannella and Andrzej Tarlecki. Mind the gap! Abstract versus concrete models of specifications. In *Proc. 21st Intl. Symp. on Mathematical Foundations of Computer Science*, volume 1113 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 1996.

[SW82] Donald Sannella and Martin Wirsing. Implementation of parameterized specifications. In *Proc. 9th Intl. Colloq. on Automata, Languages and Programming (ICALP)*, volume 140 of *Lecture Notes in Computer Science*, pages 473–488. Springer, 1982.

[SW83] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. In M. Karpinski, editor, *Proc. 11th Colloquium on Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 1983.

[Tar98] A. Tarlecki. Towards heterogeneous specifications. In D. Gabbay and M. van Rijke, editors, *Proc. 2nd Intl. Workshop on Frontiers of Combining Systems, FroCoS'98*. Kluwer, 1998.

[Tse91] T.H. Tse. *A Unifying Framework for Structured Systems Analysis and Design Models*. Cambridge University Press, 1991.

[UML97] UML notation guide, version 1.1. Technical Report ad/97-08-05, Object Management Group, 1997. Available from `http://www.omg.org`.

[Var92] P. Varnish. Proof obligations in Extended ML. Master's thesis, University of Edinburgh, 1992.

[WDC+94] U. Wolter, K. Didrich, F. Cornelius, M. Klar, R. Wessaly, and H. Ehrig. How to cope with the spectrum of SPECTRUM. Report 94-22, Technische Universität Berlin, 1994.

[Wir86] M. Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science*, 42:123–249, 1986.

[Wir90] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 13, pages 675–788. Elsevier Science Publishers B.V. (North Holland), 1990.

[Wir93] M. Wirsing. Structured specifications: syntax, semantics and proof calculus. In F. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 411–442. Springer, 1993.

[Wir95] M. Wirsing. Algebraic specification languages: an overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types – Selected Papers*, volume 906 of *Lecture Notes in Computer Science*, pages 81–115. Springer, 1995.

[Wol94] B. Wolfinger, editor. *Innovationen bei Rechnern und Kommunikationssystemen*. Springer, 1994.

[ZHR92] C.C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1992.