# Formal specifications in general, and some current research topics in algebraic specifications

Gilles Bernot
Université d'Evry - Val d'Essonne
La.M.I. (Laboratoire de Mathématiques et d'Informatique)
Cours Monseigneur Roméro
91025 Evry cedex, France
e-mail: bernot@live.univ-evry.fr

November, 1995

## 1 The choice of formal specifications

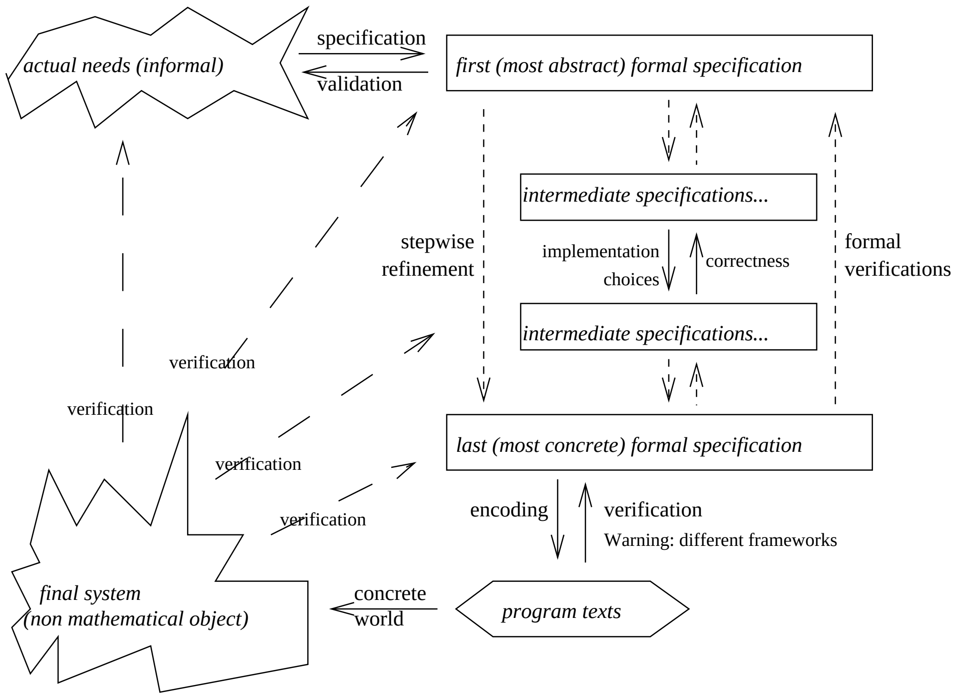
The importance of formal methods in software manufacturing is growing up. The use of formal methods becomes a sort of "label of quality" which is often considered as a guarantee of a certain level of software reliability. Of course, such an approach only takes sense if the objective itself is formally identified. Consequently, formal methods rely on some *formal specifications* which state clearly what has to be formally ensured. This requires a lot of competences and investments; it is the reason why the critical parts of software engineering projects are the best "clients" for formal specifications and formal methods.

The first role of a specification is to establish as clearly as possible *what* has to be done. It is a sort of contract to be fulfilled, which has to be known before thinking about *how* to realize it. In practice, it is often difficult to write "unbiased" specifications at each level, i.e., specifications that stay within the boundaries of this "*what*," without premature choices which mention a unnecessary "*how*" part. Most of the time, making specifications abstract enough is a strong discipline, a sort of "pedagogical" effort. However this effort is profitable, especially if the specified component is used (or reused) several times: the abstraction effort to understand the purpose of the component has not to be done several times by numerous readers.

Moreover, if we want to rigorously treat the question of software correctness, it is necessary to associate a rigorous *semantics* (mathematically defined) to each statement (i.e., to the *syntax*) of a specification. Formal specifications define *unambiguously* what the correctness of a program signifies and they are indeed the only way to have a rigorous definition of correctness. Consequently, formal specifications must be used if we want to consider "entirely proved programs," "zero-default softwares," etc. From a logical point of view, the notion of correctness of a program, without a formal specification of it, is a non-sense.

**Slogan:** *To prove a theorem, it must be rigorously stated. To reach software correctness, it must be formally specified.*

Indeed, the fact that formal specifications are the only way to rigorously define software correctness is not the main motivation. In practice, for the crucial parts of a software project, formal specifications oblige the specifier to treat a lot of particular cases that would have been forgotten, or ambiguously specified otherwise. They also facilitate a “mutual validation” between two texts written according to formal syntactical rules. This mutual validation is not necessarily done between a specification and a program; it is often done between two formal specifications, one of them being a refinement of the other. The use of formal specification languages gives a deeper understanding of the specified properties. Thus, erroneous or divergent interpretations by the various development teams are avoided, or at least early identified. Formal specifications allow to perform the *stepwise refinement process* (from the most abstract formal specification to the most concrete one) in a fully reliable manner since all verifications can be made in a purely mathematical world.



It is not so easy for the validation of the most abstract formal specification with respect to the needs of the clients, because the “needs of clients” are not mathematically specified (else this specification itself would have to be validated. . . ). Moreover almost the same problem arise for the verification of the final system with respect to its specifications, because a concrete system is not a pure mathematical object, nor is a program execution.

*With that respect, what is the usefulness of formal specifications ?*

As a matter of fact, we must admit that “zero-default” is unreachable because the real world is not a mathematical object, and a sceptic reader may wonder if the “beautiful mathematical corpus” of formal specifications is useful. Indeed, it is an experimental fact that formal methods (consequently formal specifications) increases software quality. Precisely, one of the advantage of rigorously treating correctness is to oblige to explicitly mention the *hypotheses* under which the system can be considered as correct (correctness of the compiler, of the hardware, form of the environment, etc.).

## 2 Formal specifications: several frameworks

It becomes now evident that a "universal formal specification framework" is a myth, similar to the old myth of a "universal programming language." Even if the theory proves that, more or less, all programming languages have the same power (equivalent to the Turing machine), the practice shows that certain programming languages are better suited for certain applications (e.g., languages with resolution for data base requests, functional languages for symbolic treatments, etc.). To merge all aspects of programming languages into a unique programming language seems to be hopeless: the resulting languages are far too rich and complicated to be efficiently usable. The situation is even worth for formal specifications. Here again, a sceptic reader may affirm that there does exist a universal specification language, which is the mathematics itself (say logic to be more precise). Unfortunately, the language of mathematics is far too rich to be usable. A good formal specification language should also provide the specifier with a lot of easy-to-use techniques and tools, which help to write specifications, modularize them, prove properties, generate test cases, facilitate prototyping, and so on. Such languages are of course more restrictive than the mathematical language, but they offer specific advantages for validation, verification or whatever. This position between the "mathematical approach to avoid ambiguities" and the "too rich mathematics" motivates a rather restrictive view of what is a formal specification theory. It is commonly said that a formal specification theory should offer :

- a rigorous *syntax* strongly defining what a specifier is allowed to write to obtain a specification
- mathematically defined *semantics* describing (all) the model(s) associated to a given specification
- a rigorous syntax to define well formed *formulae* and a (not necessarily complete) set of rules allowing to prove if certain properties are satisfied by (all) the model(s) of a given specification.

The role of the semantics is to establish and justify the soundness of the rules used to perform proofs with respect to a given specification. Consequently, in practice, a specifier is not obliged to understand all the "so complicated mathematical considerations" involved by semantics; it is sufficient for a specifier to have a good intuitive idea of what his or her specification means, provided that he or she is able to perform proofs in order to check the required properties. Of course the *soundness* of the set of rules used to perform proofs is crucial; it should have been established once and for all by the author(s) of the specification theory, according to the semantics.

According to this restrictive view of "formal specifications" we may distinguish two classes of formal specification frameworks. They can be called respectively *model oriented* and *property oriented* (or *declarative*).

In the model oriented approaches (VDM [Daw91][Jon86], Z [Spi89], B [Abr]...) the specifier builds a *unique* model, from a lot of built-in data structures and construction primitives that the specification language offers. Then, a program is *correct* with respect to the specification if it has the "same behaviour" than the specified model. It is important, in the software development process, to always keep in mind that the final software should not necessarily follow the same construction than the specification, nor the same data structures.

In the property oriented approaches, the specifier gives first a list of *functionality names* and by default there is an infinity of models that provide, in different manners, a functionality for each name. Next, the specifier declares several properties (i.e., formulae, which are often called "axioms" as they have not to be proved; they are simply required). Among all the previously mentioned models only a few of them satisfy the required properties; all the other models "do not satisfy the specification" and are discarded. Then, a program is *correct* with respect to the specification if it provides the users with all the declared functionality names and if the way it manages its internal data structures in order to perform those functionalities defines a model that satisfies the specification. A specification is *consistent* if there exists at least one model that satisfies this specification.footnoteConsistency is not decidable in general but there are usually usable sufficient conditions to ensure consistency. Also, in the so-called "initial approach," the semantics is restricted to a particular class of models (all isomorphic) where two values are different by default, except if the axioms impose an equality. This class of model always exists if the syntax is restricted to a particular kind of axioms which resemble, more or less, to Horn clauses.

Last, but not least, it is useful to make a clear distinction between a *formal specification theory*, a *formal specification language* and a *formal specification environment*:

- A formal specification theory is a mathematical definition of a syntax and its corresponding semantics, as well as an associated set of deduction rules, and possibly some other results specific to the considered approach. It should be clear that such a theory is not sufficient to start writing real sized specifications. For instance, the syntax should be understood as an "abstract syntax," where details such as definition of keywords, infix or mixfix notation of terms, etc., are not dealt with. Moreover a lot of semantic constraints are not considered (for example, often the treatment of modularity aspects may be not included).
- A formal specification language defines a "concrete syntax" and it fully defines all semantic options such as the treatment of modularity or the treatment of parameterization. It may also restrict the specification theory in order to define certain primitives, to enrich the proof rules, or to reduce the semantics (a typical example is to restrict the syntax in order to follow an initial semantics). It may also provide the specifier with primitives for the specification development, etc.
- A formal specification environment is a software system, usually attached to a formal specification language, which provide the specifiers with useful tools. Among many possible tools, let us mention guided editors dedicated to the concrete syntax, proof assistants, test case generators, prototyping facilities, code generators (ML, ADA, LISP, C...), libraries of ready-to-use specification components, etc.

## 3 Algebraic specifications

Property oriented approaches are often based on abstract data types and the most "popular" approaches are based on *algebraic semantics.* Algebraic specifications denote indeed a lot of different specification languages (LARCH [GH86], ASL [SW83][Wir86], PLUSS [Gau92], CLEAR [BG80][San84], OBJ [FGJM85], ACT-ONE/ACT-TWO [EM85]...) and an incredible number of different specification theories. The paradigm of algebraic specifications is rather a way of thinking semantics, with a common set of mathematical tools (based on category theory) to establish basic properties of the proposed specification languages, and with a syntax mainly based on equalities. Algebraic models are usually set of values, in most of the approaches each value has one or several *types* and there are "operations" or "functions" which work on those values according to their type(s) [GTW78][EM85]. As we can imagine, in such a context "time issues" are rarely dealt with. A common opinion is that algebraic specifications should be used to describe functional aspects, in complementarity with other frameworks (such as temporal logics, Petri nets or transition systems) to specify real time systems, systems with parallelism or concurrency. On the other hand, algebraic specifications are fully adequate to deal with modularity, stepwise refinements, exception handling, typing issues, reusability, etc. (i.e., more or less all the classical problems of software engineering with respect to sequential programs).

From an academic point of view, algebraic approaches of specification have the great advantage to formally answer the question “What is *correctness* for programs ?” using well established mathematical tools and they serve as reference to build and study fully reliable software development and verification methods. From an industrial point of view, algebraic specifications may be a reference in the near future. The wide variety of algebraic specification theories and languages will help to face the complexity and the variety of the requirements to be specified in huge size softwares. Several specification languages or dialects will probably be used in the future, each of them being dedicated to specific areas (nuclear plants, public transportation, telecommunications, hardware, etc.) For the time being, the wide variety of academic formal specification languages is rather understood as a handicap for formal specifications instead of a flexibility advantage. People need to know how to compare formal specification languages, and academic people lack of easily usable tools with that respect. There are many and many academic tools to support algebraic approaches, but they often remain prototypes with poor interfaces. Moreover, existing reliable tools are often dedicated to a specific approach of algebraic specification.

Nevertheless, a lot of such academic tools are very convincing. For example:

- automatic or computer aided prototyping from algebraic specifications
- incremental integration of modules where stubs and drives are automatically derived (or computer aided) from algebraic specification modules
- ADA, C, LISP, ML automatic code generation (or computer aided) from algebraic specifications; for example the MIT project LARCH gives well elaborated tools.
- since formal specifications follow entirely established syntactic rules, we can a priori design automatic test selection tools from the specifications (black-box/functional testing) in a similar manner as automatic test selection tools exist from the program text (glass-box/structural testing). Such an approach as been explored in France for algebraic specifications [Ber91][BGM91] and gives powerful test data sets [BGLM93]:
  - a case study on a module extracted from a nuclear plant system gave us better results than classical structural testingfootnoteapproximatively 99.98% against 84%.).
  - a case study on overspeed alarm for a subway train automatically focused the test data sets on critical combinations of events and several tests have been selected that were never proposed by the experts of testfootnoteabout 3 previously unknown bugs have been revealed (3 is a big number when human life is under consideration).
- more generally, almost all well known tools working on the program texts should give rise to similar tools for formal specifications, taking advantage of their fully established syntax and semantics.

## 4 Miscellaneous issues

This section contains some issues that, I believe, may have some general interest, and some other ones that I find exciting (perhaps because they belong to the current research subjects of my team). Don’t worry if there seems to be no logical continuity in the text.

## 4.1 How to validate an abstract specification

While correctness proofs can be considered to verify a program (or an intermediate specification) with respect to its (higher level) specifications, correctness proofs for the most abstract specification (see the figure) are impossible by nature. One could *prove* the correctness of the abstract specification only with respect to a *formal* definition of "the needs of the clients," but precisely, such a formal definition (possibly partial) would be the most abstract specification. Consequently, the validation stage must rely on informal, "good sense" arguments. For this reason, it is of first interest to use a formal specification language whose formulae are very closed to the direct intuition of the readers and the direct will of the clients. Consequently, the semantic side of the specification language is a key point in the validation process. For instance, it should allow the specifier to clearly state the *required properties* and not any *way to realize them.* To give a simple example, to specify an operation that sorts a sequence of elements, it is unadequate to provide an algorithm at the most abstract level. Such an algorithm should be considered only after some refinement. A good abstract specification of a sorting operation should simply say that the result is a permutation of the input, and is sorted. *These* are the only properties wanted by the users, and they will have to be verified (e.g., proved) from the chosen implementing algorithm.

To validate the most abstract specification, another sensible way would be to "practice" it and to check if it corresponds to the needs. In other words, it means to experiment a correct realization of the specification... Of course, it is not realistic, from the software engineering point of view, to wait until the final realization is done before validating the first abstract specification. It is the reason why the ability to *prototype* an abstract specification (even partially, as it is often the case) is also a major consideration. Unfortunately, it is quite evident that prototyping is, to a large extent, antagonistic to the abstract specification style mentioned before. Consequently, it may be necessary to perform a refinement step before to generate a prototype. Nevertheless, prototyping may be, sometimes, an acceptable argument to restrict the expressive power of certain specification languages. Let us note that such considerations are not necessarily as restrictive as it may seem: fortunately, prototypes are not always extracted from a specification via elementary tools such as the Knuth-Bendix algorithm... Moreover, partial prototypes (which do not entirely realize the specification) are still useful; they have the great advantage to "give an idea" of the semantics of a specification.

Specifications cannot always be (easily) prototyped, and fortunately, prototyping is not the only way to experiment them. When specifying, there exist many properties which we do not include into the specification axioms, but that should be ensured, according to the intented behaviour. Consequently, another way to validate an abstract formal specification is to (try to) prove those additional properties that "should be true if the specification does what we believe it does." The success or the failure of such proofs can induce some modifications of the specifications very early, which considerably reduces the number of wrong design decisions. Consequently, a sound calculus (if not complete) is a major criteria to facilitate the validation stage. Of course, when the semantics offer a sufficient expressive power (to allow a very abstract specification style as mentioned before), there are often a lot of undecidable properties. However, in practice, the additional properties should be immediate consequences of the specification, and if the proof fails, it is a good hint to modify the specification (even if the calculus is not complete).

## 4.2 How to decompose a specification into manageable pieces

The decomposition of specifications is not only a syntactical question. For programming languages, the decomposition of a software into smaller components is based on encapsulation principles, separation of variables, etc., which allow separate developments. Similarly, the decomposition of a specification into smaller components has semantic impacts, since it should reflect the program component semantics, allow a better modularization of proofs, offer reusability issues, be compatible with stepwise refinements, etc. Consequently, the definitions of "how to structure specifications" are often deeply intricated with the underlying specification theories.

Roughly, the decomposition choices can be guided by the data structures underlying the specification, or by a classification of the main tasks to perform. The first approach is mainly based on a decomposition into *modules* (e.g., one module per data type). The second approach is often based on a decomposition into objects, or *object types*.

- In a modular approach, a specification is decomposed into smaller modules. A module mainly states what the underlying data structure is able to do: which operations are available, and their properties. A module can use other modules. Used modules usually specify lower level data structures which are necessary to specify the type(s) of interest. Often, modules are organized into a hierarchy (consequently without cycles with respect to the "use" relation between modules). In practice, a module is indeed a collection of functions which work on a particular data type of interest (the heuristic "1 type $\leftrightarrow$ 1 module" is often followed).
- In an object oriented approach, a system is viewed as a community of objects which cooperate to perform, at the end, what the system is supposed to do. This "cooperation" between objects can induce cycles, as opposed to the "use hierarchy" of a modular approach. An object is a unit of structure which has an identity and an internal state, and which is capable of performing methods, cooperating concurrently with other objects. Objects which share similar structures and behaviours in a system are often grouped in a class, and this common behaviour defines an object type. There can be inheritance relations between object types; they mainly allow to efficiently add methods or modify method behaviours.

## 4.3 How to get a program from a formal specification

Assuming that a first abstract formal specification of "what the system is supposed to do" is available, it has to be *refined* in order to incrementally define more concrete specifications and to make precise the details of implementation. The intermediate specifications reflect more and more implementation (i.e., "how") choices, until the most concrete specification is detailed enough to extract a program.

An elementary refinement step is often called an abstract implementation. Starting from a (more) abstract specification, it consists in specifying a realization of each operation, using several lower level specifications. The resulting (more) concrete specification has to be verified with respect to the (more) abstract one. Then, the used lower level specifications can be refined themselves.

## 4.4 How to test a program against a formal specification

Assuming that a formal specification is available, one can formally study the verification of a program with respect to its specification. While proof theories are widely investigated, testing theories have not been extensively studied. However, in practice, when big softwares are involved, a complete proof is often impossible, or at least it is not realistic because it would be too costly. The crucial properties of the program under test should be proved, but several less critical properties can be checked by testing.

Most of the current methods and tools for software testing are based on the structure of the program to be tested ("white-box" testing). Using a formal specification, it becomes possible to also start from the specification to define some testing strategies in a rigorous and formal framework. These strategies provide a formalization of the well known "black-box" testing approaches. They have the interesting property to be independent of the program; thus, they result in test data sets which remain unchanged even when the program is modified. Moreover, such strategies allow to test if all cases mentioned in the specification are actually dealt with in the program.

The main advantage of this approach is to formally express what we do when testing. It allows also to modelize cases where some properties have been proved. Moreover, for algebraic specifications, it is possible to automatically select test data sets from a structured specification. LOFT [BGM91] is a system developed in PROLOG, which allows to select a test data set from an algebraic specification and some hints about the chosen testing strategy. The testing strategies are reflected by hypotheses. The hypotheses play an important role: they formalize common test practices and they express the gap between the success of the test and the correctness. The size of the selected test sets depends on the strength of the hypotheses.

The underlying idea is rather simple: to get a test data set out of a specification and a set of reasonably strong hypotheses, it is sufficient to try to prove (a' la PROLOG) that the hypotheses imply the specification. Of course, the proof fails (else the hypotheses are "I assume that my program is correct," which is obviously too strong). The nodes where the proof fails are indeed sensible cases where the program could be wrong... they constitute valuable test cases. This approach is more or less what LOFT does.

## Concluding convictions

The main advantage of formal specifications is to turn specifications into formal texts that can be treated with powerful tools or methods. These tools or methods are based on rules which are rigorously established with respect to the corresponding formal semantics.

Algebraic specifications are well suited to deal with classical problems of sequential software engineering (modularity, stepwise refinements, exception handling, typing issues, reusability, test, etc.); they are less usable for real time or parallelism issues. However, even if classical algebraic specifications were more or less limited to the description of abstract data types in the past, there are dynamic researches on, precisely, "dynamic types" and object orientation. For instance, the specification of software and hardware systems with hidden memory states becomes possible, and produces rather easily readable specifications [AB95]. These researches seem to make algebraic specification able to specify temporal properties, using some kind of temporal logics to perform proofs.

My belief is that in the near future, formal specification languages will be chosen (and mixed together) with the same facility than programming languages: the choice should depend on the problem (or part of problem) under consideration.

## References

[AB95] Aiguier M., Bernot G. : "*Algebraic semantics of object type specifications.*" In Information Systems Correctness and Reusability, Selected papers from the IS-CORE workshop, September 1994, World Scientific Publishing, R.J.Wieringa R.B.Feenstra eds, 1995.

[Abr] Abrial J-R. : "*Assigning programs to meanings.*" Book to appear.

[Ber91] Bernot G. : "*Testing against formal specifications: a theoretical view.*" Proc. of the International Conference on Theory and Practice of Software Development (TAPSOFT'91 CCPSD), Brighton U.K., April 1991, Springer-Verlag LNCS 494, p.99-119.

[BG80] Burstall R.M., Goguen J.A. : "*The semantics of CLEAR, a specification language.*" Advanced Course on Abstract Software Specifications, Copenhagen, Springer-Verlag LNCS 86, p.292-332, 1980.

[BGLM93] Bernot G., Gaudel M.-C., Le Gall P., Marre B. : "*Experience with Black-Box Testing from Formal Specification.*" 2nd international conference on Achieving Quality in Software (AQuIS'93), Venice, Italy, October 1993.

[BGM91] Bernot G., Gaudel M.-C., Marre B. : "*Software testing based on formal specifications: A theory and a tool.*" Software Engineering Journal (SEJ), Vol.6, No.6, p.387-405, November 1991 .ALSO LRI Report 581, Université de Paris XI, Orsay, France, June 1990.

[Daw91] Dawes J. : "*The VDM-SL reference guide.*" Pitman, 1991.

[EM85] Ehrig H., Mahr B. : "*Fundamentals of Algebraic Specification 1. Equations and initial semantics.*" EATCS Monographs on Theoretical Computer Science, Vol.6, Springer-Verlag, 1985.

[FGJM85] Futatsugi K., Goguen J.A., Jouannaud J-P., Meseguer J. : "*Principles of OBJ2.*" Proc. 12th ACM Symp. on Principle of Programming Languages, New Orleans, january 1985.

[Gau92] Gaudel M-C. : "*Structuring and modularizing algebraic specifications: the PLUSS specification language, evolution and perspectives.*" Proc. of the 9th Symposium on Theoretical Aspects of Computer Science (STACS), Cachan, France, February 1992, Springer-Verlag LNCS 557, p.3-18, 1992.

[GH86] Guttag J.V., Horning J.J. : "*Report on the LARCH shared language.*" Science of Computer Programming Journal, Vol.6, No.2, p.103-134, 1986.

[GTW78] Goguen J.A., Thatcher J.W., Wagner E.G. : "*An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.*" Current Trends in Programming Methodology, ed. R.T. Yeh, Printice-Hall, Vol.IV, pp.80-149, 1978. (Also IBM Report RC 6487, October 1976.)

[Jon86] Jones C.B. : "*Systematic software development using VDM.*" Prentice Hall, 1986.

[San84] Sannella D. : "*A set-theoretic semantics for CLEAR.*" Acta Informatica, Vol.21, p.443-472, 1984.

[Spi89] Spivey J.M. : "*The Z notation: a reference manual.*" Prentice Hall, 1989.

[SW83] Sannella D., Wirsing M. : "*A kernel language for algebraic specification and implementation.*" Proc. of the Intl Conf. on Foundations of Computation Theory (FCT), Borgholm, Sweden, Springer-Verlag LNCS 158, p.413-427, 1983.

[Wir86] Wirsing M. : "*Structured algebraic specifications: a kernel language.*" Theoretical Computer Science (TCS), Vol.42, No.2, p.124-249, 1986.