Formal Specifications and Algebraic Specifications

Gilles Bernot

Laboratoire de Mathématiques et d'Informatique
Université d'Evry – Val d'Essonne
Bd des Coquibus
F-91025 Evry Cedex
FRANCE
e-mail: bernot@lami.univ-evry.fr

– Position Paper –
Proc of *The 7th International Software Quality Week*, San Francisco, 17-20 Mai 1994

1 Introduction (Formal Specifications in General)

In this paper I will adopt a rather restrictive definition of "formal specifications". I will say that a specification framework is "formal" if it contains :

- a rigorous *syntax* entirely 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" *properties* 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. I believe that 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 framework, according to the semantics.

One of the main advantages of formal specifications is that they *never* contain an *ambiguity*. They entirely define 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 fault softwares", etc. Nonetheless, the idea of using formal specifications as absolute correctness reference is not as good as it seems. When an inadequation between a formal specification and a program supposed to implement it is found, the practice reveals that almost half of the time (say 1/3) it comes from an error in the specification (and not necessarily a bug in the program). In practice, the first advantage of formal specifications is to oblige the specifier to give a complete specification (including exceptional or rare cases) and to 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.

We meet here another important criteria for formal specifications: the ability to go from the higher level specification to the program code by stepwise refinements (each step is often called an *abstract implementation*). Each step should be an elementary step where mutual correctness proofs are not too difficult to handle; it is possible to have a great number of such elementary steps. Let us remark that there is a classical problem with formal specifications: the higher level formal specification cannot be validated with respect to an external specification since it can only be compared with something unformal (e.g. "what we have in mind"). It is only possible to obtain a partial validation, for example we can try to prove a property that "should be ensured if the specification does what we believe it does" (if the proof fails, the specification is probably wrong or uncomplete).

Lastly, I strongly believe that there is no "universal formal specification framework." In the same way that it is now well known that there does not exists a universal programming language (we have to choose a programming language according to the intended application), we have to choose a formal specification framework according to the problem under consideration. For some applications, it seems preferable to use in the same time *several* specification frameworks in order to give several point of view on the same object (e.g. functional requirements vs. real time requirements). Consequently I will not affirm that algebraic specifications must always be used...

2 Algebraic Specifications

According to the restrictive definition of "formal specification" developed in the first section, we may distinguish two classes of formal specification frameworks. The first one can be called "model oriented" and is more widely used[1] than the second one which can be called "property oriented" or "declarative."

In the model oriented approaches (VDM [Daw91][Jon86], Z [Spi89], B [Abr94]...) 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 behavior" 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 (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 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. Also, a specification is *consistent* if there exists at least one model that satisfies this specification.[2]

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

[1] At least in Europe

[2] Consistency is not decidable in general but there are usually usable sufficient conditions to ensure consistency.

of different specification frameworks (LARCH [GH86], ASL [SW83][Wir86], PLUSS [Gau92], CLEAR [BG80][San84], OBJ [FGJM85], ACT-ONE/ACT-TWO [EM85]...). 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. My 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 but I believe that, for the moment, they lack of easily usable tools. 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 testing[3]).
  - 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 test[4]
- 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.

[3] approximatively 99.98% against 84%.

[4] about 3 previously unknown bugs have been revealed (3 is a big number when human life is under consideration).

Concluding remarks:
The main advantage of formal specifications (with the restrictive definition of first section) 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, etc.); they are less usable for real time or parallelism issues. My belief is that in the near future, formal specification languages should 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

[Abr94] Abrial J-R. : "*Assigning programs to meanings.*" To appear, 1994.

[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, Universite' 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.