1 The Role of Formal Specifications

Marie-Claude Gaudel[1] and Gilles Bernot[2]

[1] Université de Paris-Sud, LRI, CNRS UMR 8623, Bâtiment 490, F-91405 Orsay Cedex, France
[2] Université d'Évry, LaMI, CNRS EP738, F-91025 Évry Cedex, France

Abstract. This introductory chapter aims at stating the context and the motivations of the rest of the book. The first section is a brief general reminder of the role of specifications in the software development process. Important concepts such as abstraction, refinement, validation, and verification are introduced informally. The second section gives a characterization of formal specifications, sketches a classification, and discusses the possibilities that they bring for software development. Section 3 is devoted to the use of formal specifications for requirement engineering and validation. Section 4 addresses the notions of refinement and verification. Section 5 discusses what kind of tools can be developed on the basis of formal specifications.

1.1 The role of specifications in software development

Software (and hardware) systems have to be developed following rigorous guidelines to have a chance of being dependable, finished on time, and easy to maintain. The aim of *software engineering* is to define criteria to evaluate software-intensive systems, as well as methods and techniques to develop them and satisfy these criteria. The search for these criteria, methods, and techniques requires a study of the characteristics of such systems and their development process.

The software development process can be seen as the construction of a sequence of more and more detailed descriptions of the software under development, leading to a final set of documents which contains an executable program and its documentation. This view is clearly oversimplified since it ignores the essential role of backtracking and iterations in such a process [Hum90, GMSB96], but it is sufficient for the purpose of this introductory chapter.

At every stage of the development process but the last one, the core of the description document is a *specification* of the future software, namely a definition of *what* it must do, without a complete description of *how* it will do it.

Because it plays a special role, it is interesting to distinguish the earliest specification document: we will call it the *global specification*, while the other ones are called intermediate or detailed specifications.

Clearly, the concept of specification is a cornerstone of software development: the global specification is the basis of the agreement between the developers and the users (some authors see it as a "contract"); global and intermediate specifications are essential for communication of precise information between the developers.

At each stage of the development, the current description of the future system must be checked against inconsistencies and omissions.

Moreover, it must be *validated* if it is the first one, or *verified* with respect to the previous one. Namely, it is necessary:

- to validate the global specification with respect to the requirements, the needs of the clients;
- to verify every intermediate specification with respect to the previous specification;
- to verify every piece of program with respect to its detailed specification;
- to verify the integrated final system with respect to its global specification.

Validation aims at establishing that the future system will fit for its operational mission. Verifications aim at establishing that it will satisfy the global specification. Validation, verification, consistency, and completeness checks are crucial activities since it has been recognized for a long time [Boe81] that the later a fault is discovered in the development process, the more expensive it is to correct.

The ability to use the specifications effectively for verification and validation is a major issue. For example, depending of the kind of developed system, a specification technique should provide some support for prototyping, correctness proofs, elaboration of test data, and failure detection when running these tests.

As said above, the role of the global specification is to establish as clearly as possible what has to be done. This must be understood and stated before deciding 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 the "*what*," without premature choices which mention unnecessary "*how*" aspects. However this effort of abstraction is profitable:

- *A priori*: before the software under consideration is finished, an abstract specification is a powerful reference document for discussions and elaboration of the design; besides, abstraction prevents local design choices which could result in a poor global design;
- *A posteriori*: a sequence of specifications, progressing from abstract to concrete, provides a trace of the design activity and makes easier the reuse, evolution, and maintenance of the software.

The transition from a specification to a more detailed one is often called a *refinement*. Most specification techniques provide some support for such refinements and the corresponding verifications. Verification may cause some iterations on the refinement until its result, namely the more detailed specification, is satisfactory with respect to the original specification.

Specifications are more abstract than the corresponding programs, but in many cases they are big enough to require some way for mastering size and complexity when writing them. As for programming languages, this is mainly provided by decomposition and modularity principles (see Chapters 6 and 8 of this book), or, more recently, by object-orientation (see Chapter 12). However, it is worth to note that the structure of a specification is mainly dictated by understanding the role of the future system. It is not the case of the structure of the implementation, which may be very different due to considerations such as efficiency, security, or reuse [FJ90, Gau92]. Thus, elaboration of the software architecture [SG96] may lead to a structure different from the organization of the specifications.

Like the choice of a programming language, the choice of a specification technique depends on the kind of system to be specified. For certain projects, it can even depend on the components to be specified: for example, certain components may require to specify real time aspects, while some other parts may require a careful specification of complex data structures without time constraints. A specification technique which would deal with all the possible aspects of all the possible systems would have a good chance to be too complex to be usable. This justifies the existence of a variety of languages and methods, each of them emphasizing particular aspects of some application domain and some class of target programs. As a consequence, in some cases it may be useful to write several specifications using different specification techniques for some component, in order to enlighten different aspects.

In the current state of practice, specifications are often written using diagrams or tables, enriched by comments in natural language [Dav93]. These diagrams and comments are the major media for dialog between the actors of the development. However, they often admit several interpretations. This plurality of interpretations results from *ambiguities* in the used notations. Such ambiguities of informal specifications do not necessarily prevent from producing software of good quality. However, they may raise some problems, especially when checking, validating, or verifying.

This book presents a class of *formal specification* techniques. Formal specification techniques make it possible to remove any ambiguities in the expression of the specifications of a program, in order to provide a sound basis for checking, validating, and verifying.

1.2 Formal specifications

A specification written in a formal specification language defines rigorously all the acceptable behaviors of the specified system. Besides, it is possible to perform some reasoning on the specification and its refinements.

A formal specification technique must provide at least three well defined facets: a *syntax*, some *semantics* and an *inference system*.

- The syntax exhaustively defines what a specifier is allowed to write to obtain a specification, as it is the case for programming languages. Thus the text of a formal specification has a structure which makes possible powerful computer aided treatments. In particular, a specification contains properties which are written as well formed formulas of some logic (these properties are often called axioms or sentences in this book).
- The semantics describes the models associated to a given specification. A model is a mathematical object which defines the behaviors of the acceptable realizations of the specification. "Acceptable" implies in particular that each model satisfies the properties of the specification. The role of the semantics is precisely to avoid ambiguities.
- The inference system serves to define the deductions that can be made from a formal specification. These deductions allow some formulas to be derived, which are consequences of the properties listed in the specification. Such derivations are mechanically checkable as developed in Section 4. So, the inference system can help to partly automate theorem proving, functional testing, prototyping, verification of refinements, etc.

This classical view of formal specifications leads to distinguish (at least) two classes of formal specification techniques. They are called respectively "model oriented" and "property oriented" (or "constructive" and "declarative").

In the *model oriented* approaches (such as VDM, Z, or B) the specifier builds a *unique* model, from a choice of built-in data structures and construction primitives that the specification language offers [Jon90], [Spi92], [Abr96]. Then, a program is *correct* with respect to the specification if the functions it provides have the *same behaviors* than in the specified model.

In the *property oriented* approaches, the specifier declares first a list of "function names" and by default there is an infinity of models that provide, in all the possible ways, a function for each name. Next, the specifier states several properties (i.e., formulas, which are often called "axioms" as they have not to be proved: they are 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 a specification if it provides all the declared function names and defines a model that satisfies the specification. It is this class of specification techniques, also called *algebraic specifications*, which is studied in this book.

Among other classes of formal specification techniques, there are operational specifications. Starting from a set of elementary actions, these techniques describe the computations, i.e., the sequences of actions, that the system can perform. Some representatives of these techniques are Petri nets

[Rei85] and process algebras [Hen88]. Extensions of algebraic specifications on this direction are presented in the Chapter 13 of this book.

Formal specifications define unambiguously the correctness of a program with respect to its specification. They are indeed the only way to have a rigorous definition of correctness. Consequently, formal specifications must be used if correctness proofs are foreseen for some verifications. From a logical point of view, the notion of correctness of a program, without a formal specification, is a nonsense. Nevertheless, making use of formal specifications is a demanding process and should be suitably targeted. There are many cases where formal specifications would be a mere luxury.

In most projects where formal specification were used, informal specifications and formal ones were mixed. Some components and steps were formalized, some others not. The decision to use formal specifications mainly depends on the *criticality* of the component, in term of consequences of a fault (human lives, cost) and of the complexity of its requirements or of its development.

There are also some aspects of the development process which are beyond the scope of formal development methods and correctness proofs [Gau95]. The figure below shows the boundary of what can be formal in software development. The right hand side shows a purely formal process which goes from a high level abstract formal specification to the program. The left hand side of the figure mentions informal entities such as needs, opinions, or physical systems which belong to the real world. The formalization of such entities always induces some schematization [McD91], and is a delicate activity where some misunderstanding or error may occur.

The figure is slightly simplified since some validation against the actual needs may take place after each refinement, as developed in [Gau95], and some kinds of verification may skip some intermediate specifications.

Formal methods make it possible to perform a stepwise refinement process (from the global specification to the most concrete one) in a provably correct way. But the validation of the global specification with respect to the needs of the clients is a special case, because these needs are not formally specified (otherwise this specification itself would be the global specification and would have to be validated ...).

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 (specification against program texts, or against lower level specifications).

In the remainder of this introductory chapter we develop some of the main contributions of formal specifications in general, and of algebraic specifications in particular, to sofware development activities. In Section 3, we discuss the new possibilities that formal specifications bring to requirement

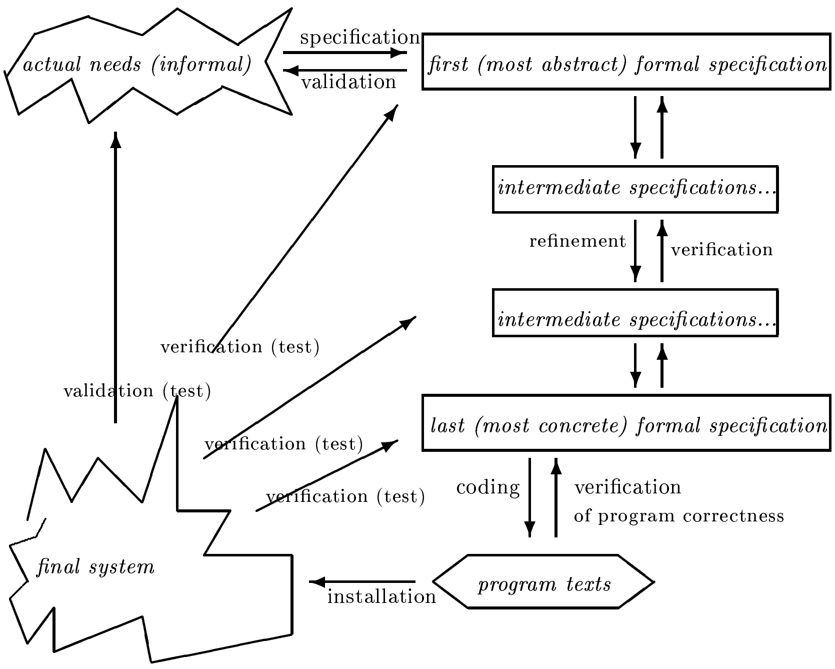

Fig. 1.1. Formal and informal aspects in software development

engineering and validation. In Section 4, we make more precise the notions of refinements and verifications. In Section 5, under the heading "mechanization of formal specifications", we briefly presents some of the development tools that formal specifications make possible. Finally, in Section 6, a quick tour of industrial use of formal specifications is presented.

1.3 Requirement engineering and validation

Even if it seems obvious, it is always useful to recall that a formal specification may be wrong. It may be wrong for two reasons: some misunderstanding of the users needs; some error in the expression of these needs. However, as said above, the notion of a formal correctness proof is meaningless in the case of a global specification since there is no formal correctness reference. Thus, the only possibility is to use "testing-like" methods. However, the great advantage of using formal specifications is that some tools exist to analyze and sometimes animate them. Such tools can be used to guide and support the validation activity.

Two classes of specification faults are of special interest in the case of formal specifications: *adequacy faults*, and *underspecifications* (for a more complete classification see for instance [Par89]). Adequacy faults arise when some of the properties expressed in the specification are in contradiction with the informal requirements. There is some underspecification when all the properties expressed in the specification are adequate, but some models of the specification correspond to unacceptable behaviors, i.e., the specification is not precise enough. A third class of specification faults are overspecifications. In these cases, the specification is too prescriptive, not abstract enough, and jumps to implementation choices which eliminate other acceptable, and maybe better, implementation possibilities.

In formal specifications, adequacy faults and underspecification may appear as inconsistencies or incompletenesses.

Inconsistency arises when contradictory properties are required. Thus there exists no implementation satisfying such a specification.

Incompleteness may (or may not) correspond to some underspecification: some properties, which are expressible in the specification language, cannot be stated as being either compatible or contradictory with the specification. Either some cases have been forgotten (underspecification) or it is just that these properties are irrelevant: it does not matter whether or not they are fulfilled by the future system.

Besides these fundamental faults, there are all the possible "typographic" faults: the cause of these faults is not a misunderstanding of the problem, but a mistake in writing the specification. It can come from a lack of attention (using one identifier in the place of another, etc.) or from difficulty in mastering the specification language. Clearly, as above, the resulting specification will be inadequate, inconsistent, or incomplete.

First, we discuss briefly how to prevent specification faults by applying some methodologic principles when developing a specification, then we give some hints on how to detect and remove such faults, making use of formal techniques.

Concerning the first kind of specification faults (those coming from a misunderstanding of the problem), an obvious methodological principle for avoiding them is to try to ensure simplicity and conciseness of the global specification. The main concept for achieving both simplicity and conciseness is abstraction [LZ74], [Neu89]. Abstraction is a key concept of algebraic specifications, the formal specification technique which is presented in this book. It is also present in most formal specification languages. However, this is not always sufficient to get abstract and concise specifications, just as modules in a programming language do not ensure that any programmer will write modular programs.

For expressing formally the required properties of a system, it is always necessary to formalize some aspects of the application domain or of the environment. This is one of the reasons why most case studies in formal specification result in apparently large specifications: it does not come from the formalization of the system itself but from the number of concepts which are necessary to the formal expression of the system functionalities. Here, as elsewhere in software engineering, structure and reuse improve the situation.

By reuse we mean reuse of theories: mathematicians do not reinvent Peano's axioms each time they use natural numbers. In computer science, a good example is the specification of compilers where accumulated knowledge is sufficient to guide and facilitate any new specification. Another less classical example of reusing pieces of formal specifications specific to an application domain is discussed in [GD90].

As indicated above, the only possible methods for validation are "testing-like" methods, in the sense that these methods can provide some evidence that the specification is wrong; they can only increase confidence in the claim that it is satisfactory. However, thanks to the deduction possibilities of formal techniques, these methods consist, most of the time, of proving or refuting consequences of the specification.

More precisely, given some conjectures (which play the role of "test data") that should be consequences of the formal specification, a theorem prover supporting the inference system of the specification technique is used to try to prove them. In [GG90] this technique is called "theory containment". Similarly, some properties which must not hold can be refuted. In [Rus93], the properties to be proved or refuted are called "challenges". This method is very powerful for detecting adequacy faults. The main difficulty is the invention of pertinent challenges which often requires a very good knowledge of the application domain.

For some formal specification languages, there is a possibility of detecting inconsistencies via some specific tool: for instance, such tools exist for algebraic specifications with equational or positive conditional axioms (see Chapter 10 of this book). It is also possible to give sufficient conditions for (some adapted notion of) completeness, which are mechanizable. *Sufficient completeness* [LG86] is such a criterion for algebraic specifications. As indicated above, the fact that a specification is consistent and complete does not mean that it is adequate. However, inconsistency reveals the presence of a fault, and incompleteness may correspond to some underspecification.

1.4 Refinements

Once a global formal specification has been stated, it has to be *refined* in order to incrementally obtain more and more concrete specifications and to make the implementation more and more precise. Each successive intermediate specification adds some implementation choices, until the last one is detailed enough to be easily translatable into a program.

In algebraic specifications, elementary refinement steps are called abstract implementations. A classical example is the representation of an abstract data type, such as set or collection, by another one, closer to a programming language, such as list or array. Another example is the choice of a precise recursive specification of a sorting algorithm, for instance a quicksort, to implement an abstract sorting operation, originally specified as returning any ordered permutation of a list. The specification of an abstract implementation consists of the definition of an abstraction function from the "implementing" data type into the "implemented" one, and the specification of a realization of each implemented operation, using the implementing data type and its operations.

The resulting (more) concrete specification must be correct with respect to the original specification, and this must be verified.

The definition of an adequate notion of *implementation correctness* is a crucial point in a formal specification technique. It should coincide with current practice and accept or reject the same realizations. But capturing a universal notion of implementation correctness turns out to be far from obvious. Different respectable definitions of implementation correctness may coexist for the same technique, depending on the application domain or on the operational context. A well-known example is the Lotos language [Bri89], and more generally process algebras.

A key point in such a definition is the fact that the implementation may be very different from the original specification, provided that these differences are not observable. Thus notions such as *observable equivalences* or *behavioral equivalences* play an essential role in such definitions. It is important to note that these notions are characterized by the kind of observations which are performable. They depend on the developed system and its environment. Thus the observability of the future system must be specified, or at least taken into account, in the verification activities. For instance, in the case of algebraic specifications, observations can be restricted to some observable data types, or to the results of some observable operations, or even to the fact that some observable formula holds or not.

To make a long story short, in almost all formal techniques the correctness criteria require that a refined specification "satisfies observationally" the properties stated in the original specification, the notion of observational satisfaction being the core of the correctness definition. These points are developed in Chapter 7.

The problem of verifying implementation correctness is made harder by the fact that in the development process, a specification evolves in two ways.

First, it is developed in a "horizontal" way, in the sense that the abstraction level is constant (and high). Most specification techniques provide structuring facilities. In this case the specification is developed piece by piece (or more likely as a collection of components which are written more or less concurrently). The result of this horizontal development is a structured specification, made of related components. *The aim of horizontal development is adequacy and completeness with respect to the user needs.*

The second evolution is the refinement process mentioned above, which is often called "vertical development" because the level of abstraction is decreasing. *The aim of vertical development is efficiency and correctness of the implementation.*

These two evolutions must be compatible, i.e., the composition of refined and correct pieces of specifications must be correct with respect to the composition of the original pieces.

Consequently, the definitions of "how to structure formal specifications" and "how to refine formal specifications" are often deeply interrelated according to the underlying specification theories.

1.5 Mechanization of formal specifications

Since there are precise rules to manipulate them, formal specifications constitute good raw material for mechanized treatments. Moreover, the soundness of these treatments can be justified with respect to their semantics. This makes it possible to support advanced aspects of computer aided software design, where each tool has a deep meaning which is sound with respect to some formal semantics.

There are four main kinds of activity where the mechanization of formal specifications can play a useful role, namely *theorem proving*, *prototyping*, *code generation*, and *testing*.

Theorem proving seems to be the first natural activity when formal methods are under consideration. Proving properties of a formal specification can be useful, as said in Section 3, for validation or, as developed in the previous section, to verify refinement steps. However, even if the underlying logic of a specification technique has a complete inference system, the truth of a formula with respect to a specification is not decidable in general as the proof tree can be infinite. So, *theorem provers* rely on *strategies*, and the various existing strategies make a significant difference between their possibilities.

The less powerful tools are *proof checkers*, which are already very useful to detect human errors in reasoning. Among more powerful tools, a distinction can be made between two classes of theorem provers: General theorem provers, such as HOL [GM93] or PVS [ORS92] among many others, can be adapted to specific formal specification techniques; Specialized ones, such as LP [GG90] for the LARCH language or MURAL [BFLM94] for VDM, are specific to a formal specification technique.

A general remark is that the more expressive is the underlying logic, the more difficult is the design of effective proof strategies. Thus interesting proofs can rarely be performed completely automatically. However, as claimed in [RvH93], the user of such a theorem prover has to struggle with its implacable logic, and this is very fruitful for debugging the specification.

Theorem proving for algebraic specifications is developed in Chapters 9, 10, and 11 of this book.

Prototyping is also an activity which is facilitated by formal specifications. Of course, ad hoc simulation tools can be developed. But techniques based on logic, such as rewriting techniques (see Chapter 9) or resolution algorithms, provide powerful bases for executing specifications written in some restricted logic.

Almost all specification languages have subparts which make it possible to write *executable specifications*. This is the case for algebraic specifications, when the axioms are restricted to equational clauses. Then, the prototyping activity can be seen as a sort of refinement process whose target is to write a specification within the boundaries of this sublanguage.

If the specification is executable, it can be tested and, since there is a formal syntax, some coverage criteria for the text of the specification, similar to those existing for programs, can be defined.

Code generation from formal specifications has been successfully studied for a long time, the pioneering project in this domain being the CIP project [?]. Almost all formal specification environments have a code generator. Code generation considerably facilitates good programming disciplines and eliminates programming errors. In order to avoid problems of efficiency of the generated code, it is recommended to start from a rather detailed specification, obtained by successive refinements of the global specification, as discussed in the previous section.

However, the correctness of the translation may be difficult to state. Another approach to code generation, which avoid this problem, is to try to obtain a program directly from an abstract specification by synthesis. These methods are based on the expression of an induction principle attached to each data structure and they can be based on intuitionistic constructive logics. A classical approach consists in proving the realizability of a specification of the form: for each symbolic input I of a specified function f, there exists an acceptable output O for $f(I)$ (where "acceptable" means "that satisfies the specification"). Then, since the way to prove the property is constructive, it is possible to extract the computation of O from the proof. This computation is a correct realization of f and the program is correct by construction.

Such techniques are presented in Chapter 14.

Formal specifications can play a useful role for the testing activities as well [Bri89, BGM91, DF93]. Mainly two of the difficult aspects of testing can be aided: *test data selection* and *decision on success or failure* of a test experiment.

The test selection activity corresponds to functional testing since the choice of the test data is based on the specification, not on the code of the program under test. In the case of algebraic specifications [BGM91], test cases are extracted from each axiom of the specification and "interesting" subcases are determined by some case analysis based on an adapted resolution algorithm.

Moreover formal specifications can help the decision on success or failure of the selected test cases, since the correctness of the obtained results is defined by the specification. A formal specification provides formal "*oracle*" predicates characterizing this correctness. If the notions of *observability* in the specification and the program are sufficiently close, it becomes possible to automatically derive the oracle.

These possibilities bring interesting perspectives of intensive functional testing of critical software-based systems (see for instance [DGM93]).

Toward industrial use

The transfer of formal methods into industrial practice has been slow. There are several reasons for these difficulties. Some of them are: problems of training of the developers, lack of good tool support, lack of well-documented methodologies, and poor integration of these methods into the usual software development models.

However, formal methods are more and more used for critical software, clearly because the effort is justified and sometimes mandatory. Some certification agencies require them, and some standards recommend their use [BH94]. An interesting list of industrial projects where such methods were used can be found in [CGR93], with a discussion of the way these methods were used, and the positive and negative effects of their use. Other examples can be found in [GH90], [DGM93], [Siv96]. The proceedings of the Formal Methods Europe symposia also report practical experiences in the use of these methods (see [GW96], [FJL97], which are the last two volumes), and the numerous workshops on industrial applications of formal methods which are flourishing now.

Bibliography

[Abr96] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[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.

[BFLM94] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, and R. Moore. *Proofs in VDM, a practitionner's guide.* Springer, 1994.

[BGM91] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[BH94] J. Bowen and M. Hinchey. Formal methods and safety critical standards. *Computers*, 27(8):68–71, 1994.

[Boe81] B.W. Boehm. *Software engineering economics.* Advances in Computing Science and Technology Series. Prentice Hall, 1981.

[Bri89] E. Brinksma. A theory for the derivation of tests. In *The Formal Description Technique LOTOS*, pages 235–247. Elsevier Science Publishers, North-Holland, 1989.

[CGR93] D. Craigen, S. Gerhart, and T. Ralston. On the use of formal methods in industry – an authoritative assessment of the efficacy, utility, and applicability of formal methods to systems design and engineering by the analysis of real industrial cases. Report to the us national institute of standards and technology, 1993.

[Dav93] A.M. Davis, editor. *Software requirements, objects, functions and states.* Prentice-Hall, 1993.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specification. In *Proc. FME'93*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 1993.

[DGM93] P. Dauchy, M.-C. Gaudel, and B. Marre. Using algebraic specifications in software testing: a case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3):229–244, 1993.

[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.

[FJL97] J. Fitzgerald, C.B. Jones, and P. Lucas, editors. *Proc. FME'97, Industrial applications and strenghtened foundations of formal methods*, volume 1313 of *Lecture Notes in Computer Science.* Springer, 1997.

[Gau92] M.-C. Gaudel. Structuring and modularizing algebraic specifications: the PLUSS specification language, evolutions and perspectives. In *9th Annual Symposium on Theoretical Aspects of Computer Science (STACS'92)*, volume 577 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 1992.

[Gau95] M.-C. Gaudel. Advantages and limits of formal approaches for ultra-high dependability. In Randell et al., editors, *Predictably Dependable Computing Systems, Chapter IV-A*, Springer Basic Research Series, pages 241–251. Springer, 1995.

[GD90] D. Garlan and N. Delisle. *Formal specifications as reusable framework*, volume 428 of *Lecture Notes in Computer Science*. Springer, 1990.

[GG90] S.J. Garland and J.V. Guttag. Using lp to debug specifications. In *IFIP TC2 Working Conference on Programming Concepts and Methods*. North-Holland, 1990.

[GH90] G. Guiho and C. Hennebert. SACEM software validation. In *12th IEEE-ACM Intl. Conference on Software Engineering*, pages 186–191, 1990.

[GM93] M.J. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[GMSB96] M.-C. Gaudel, B. Marre, S. Schlienger, and G. Bernot. *Précis de génie logiciel*. Masson, Enseignement de l'Informatique, 1996.

[GW96] M.-C. Gaudel and J. Woodcock, editors. *Proc. FME'96, Industrial benefits and advances in formal methods*, volume 1051 of *Lecture Notes in Computer Science*. Springer, 1996.

[Hen88] M. Hennessy. *An Algebraic Theory of Processes*. MIT Press, 1988.

[Hum90] W.S. Humphrey. *Managing the software process*. SEI series in Software Engineering. Addison-Wesley, 1990.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.

[LG86] B. Liskov and J.V. Guttag. *Abstraction and Specification in Program Development*. MIT Press, McGraw Hill, 1986.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–60, 1974.

[McD91] J. McDermid. Formal methods: Use and relevance for the development of safety critical systems. In *Safety Aspects of Computer Control*. Butterworth/Heineman, 1991.

[Neu89] P.G. Neumann. Flaws in specifications and what to do about them. In *IEEE Intl. Workshop on Software Specification and Design*, 1989.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS, a prototype verification system. In D. Kapur, editor, *Proc. 11th Intl. Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[Par89] H. Partsch. From informal requirements to a running program: a case study in algebraic specification and transformational programming. *Science of Computer Programming*, 11(3):263–297, 1989.

[Rei85] W. Reisig. *Petri Nets: an Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[Rus93] J. Rushby. Formal methods and the certification of critical system. Report SRI-CSL-93-07, SRI International, 1993. Available at `http://www.csl.sri.com`.

[RvH93] J. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, SE 19(1):13–23, 1993.

[SG96] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an emerging discipline*. Prentice Hall, 1996.

[Siv96] T. Sivertsen. A case study on the formal development of a reactor safety system. In M.-C. Gaudel and J. Woodcock, editors, *Proc. FME'96, Industrial benefits and advances in formal methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 18–38. Springer, 1996.

[Spi92] J.M. Spivey. *The Z notation, a reference manual.* Prentice Hall, 1992.