INTRODUCTION

ON THE RELEVANCE OF FORMAL METHODS TO SOFTWARE DEVELOPMENT

Christiane Floyd

TU Berlin
Sekr. FR 5-6
Franklinstr. 28/29

D-1000 Berlin 10

Motivation

As hostess of the practice-oriented part of the TAPSOFT conference, I would like to welcome you as participant in a discussion which we wish to promote both at the conference itself and beyond it. This discussion should focus on a central theme: the relevance of formal methods for practical software work. This leads to a series of questions: Under what circumstances and for what purposes should these methods be used? What gains may be expected? What resources and tools are needed? Which claims are founded or unfounded? What issues are or are not addressed by formal methods? What other kinds of techniques are required, and how can they be combined with formal methods, if desired?

We feel that this discussion is urgent in view of the many still unresolved problems in software development and the widely recognized need for higher quality; and that it needs to be encouraged because the general tendency is for authors developing formal approaches to operate in fairly closed circles, thus reinforcing each other's views, while many potential method users are left out. Some do not see the potential benefit of such methods to their work, or they feel that they cannot articulate their needs. Others are discouraged by the amount of learning involved, or, if they are interested, they do not know how to select the method best suited to their production situation. The technical contributions as well as the generous support given to our conference by several industrial firms indicate clearly that the topic is considered highly relevant by some institutions, while other practitioners flatly state that they consider our topic too theoretical and not worth their time. Perhaps this is related to the fact that the introduction of formal methods consumes considerable resources and that the use of formal methods will inevitably create yet another kind of specialist within a given community of DP specialists, which is undesirable in organizations producing application software for their own needs only. Thus, we are mainly addressing firms producing software on a large scale and therefore able and willing to consider a sizeable extra investment in methods.

In this introduction my aim is to clarify basic concepts and sketch out some lines of argumentation so as to prepare the ground for the discussion and to stimulate your thoughts on the topics taken up in detail in the seminar lectures and colloquium papers. In these contributions, the central theme of the conference is approached from various sides:

- Several papers focus on concepts of formal semantics and their application to specific tasks in software development, such as the specification of modular programs, the derivation of programs from formal specifications and the treatment of concurrency. These papers encourage us to think of programs in abstract, mathematical terms, and emphasize program correctness, correctness being understood in a very technical sense as a property which can, and should, be proved formally on the basis of a given specification, while the relation of the specification itself to what is needed in reality remains open.

- A second group of papers present software tools to support the application of formal approaches. Typically, the tools facilitate the formulation of syntactically correct specifications; they perform consistency checks and partially automate the transition from specifications to programs or the generation of test data on the basis of a given specification. Since the use of formal approaches tends to lead to considerable extra work involving bulky documents, which have to be kept up to date over a long period, the convenience offered by such tools has a decisive influence on the usability of the underlying method. Of particular importance in practical work is the possibility of making changes conveniently and consistently in all documents.

- In other papers the scope is widened to methods for software development as a whole. They report experiences with more or less formal methods gained in projects and in introducing methods in industrial settings. We can see here that methods are not well-defined "things" which can be mechanically applied to produce predictable results; rather, the introduction of a method in any given setting involves complex processes of learning, gaining experience and method adaptation, in which the original method as presented by its author serves as an inspiration and is eventually transformed to meet the actual needs of its users. Strictly speaking, there can be no formal method for software development as a whole because software development involves other issues besides formalization.

- Finally, the scope of applicability of formal methods and the claims put forward by their proponents are critically examined; in one case, by pointing to specific problems in software development which can or cannot be aided by formalization in the widest sense; in another, by showing how all formal approaches, in the end, depend on intuitive insight, so that in software development, as in all other human activities, our basic aim must be to go along with and strengthen our own

intuition. As an essential requirement for formal methods themselves, it follows that they must be suitable as aids to our intuition.

We hope that the various partial views presented by some of the leading authors in our field will enable you to better ascertain your position in relation to others. We would like to enable practitioners to see what the approaches offer, how they should be applied, and what their benefits and their limitations are; to enable teachers to gauge to what extent training in software engineering should be geared to such formal approaches, given their relevance; and to enable researchers to get feedback on the usefulness of their methods with a view to making them increasingly workable.

For all of us, I hope that we shall improve our understanding of how aspects that can be formalized interact with others that cannot, and how this can be taken into account by methods. Perhaps Wlad Turski had something like this in mind when he suggested "Formalism - or else?" as the title for the panel discussion.

I will devote the rest of my introduction primarily to the question of formal methods and their relevance to software development as a whole. In doing so, I have to combine the role of hostess in a controversial discussion aiming at mutual understanding with a strong viewpoint of my own, which I do not wish to hide. I will, therefore, base my arguments on my own views, without wishing to impose these views on you, and you are invited to disagree.

To clarify positions, I will attempt to sketch out a "formalist" standpoint, summarizing the key arguments of proponents of formal methods, as I know them, and a "counter-formalist" standpoint criticizing this line of argumentation. I consider myself a moderate counter-formalist. There is, however, no one coherent counter-formalist position. My own critique is based on a process-oriented view of software development, which I will contrast with the formalist position.

What do we mean by "formal"?

The underlying convictions of formalists and counter-formalists can be illustrated by examining various meanings of the term "formal" as given in the Oxford English Dictionary, which I have excerpted from a much longer, numbered list of usages:

formal

1 Of or pertaining to form, in various senses

a) pertaining to the form or constitutive essence of a thing: essential (opposed to material) .....

c) pertaining to the outward form, shape, or appearance (of a material object); also, in immaterial sense, pertaining to the form, arrangement, external qualities .....

d) Logic. Concerned with the form, as distinguished from the matter of reasoning .....

5 Done or made with the forms recognized as ensuring validity; explicit and definite, as opposed to what is the matter of tacit understanding .....

9 Marked by extreme or excessive regularity or symmetry; stiff or rigid in design; wanting in ease or freedom of outline or arrangement

b) in immaterial sense: having a 'set' or rigorously methodical character.

In my opinion, several of these meanings of "formal" are implied when different people, in varying contexts, speak about formalization in software development, without this becoming explicit. In particular, formalists and counter-formalists use this term differently. I would like, therefore, to point out some important types of differences between the various usages:

- How is form seen to be connected with matter or substance?

  In 1 a) form appears as constitutive essence, while in 1 c) it is reduced to outer appearance. Substituting "contents" for "matter" in our immaterial world of software development, I think formalists would tend to argue that the choice of formalism used to express our specifications (for example) will have constitutive influence on the contents to be expressed, while their critics will hold that the formalism chosen will influence the external appearance of these contents.

- What values are associated with issues of form?

  In 5, obeying given rules of form inspires confidence, because they are perceived as ensuring the validity of results. By contrast, in 9 form has distinctly negative connotations, such as restricting freedom in design and behaviour.

  I think that both of these views are to be found in our context, too; promoters of formal methods, in particular, are highly confident that formalization will lead to greater systems quality, while their critics usually think of design as a creative process in which an initial vision of the whole system is gradually refined and improved on the basis of a critical evaluation of preliminary proposals and in which formalisms may appear as a hindrance.

- To what extent is form considered on its own?

  Again, in 1 a) form and matter seem inextricably linked; in 1 d), however, which is the usage most directly related to what we mean by "formal" in the expression "formal semantics", form is explicitly severed from the contents of reasoning and considered on its own. While this may be fine for logic, I consider it dangerous in the development of software systems, which are not abstract games, but intended to affect, by their contents, other peoples' lives.

  Also, in 5 formal is seen to be opposed to what is a matter of tacit understanding. But all our understanding, knowing and communicating fundamentally rests on tacit understanding, which stems from the context of common action and experience. Therefore, for formal descriptions to be meaningful to people, they should not be opposed to, but explicitly rooted in, what is a matter of tacit understanding. In software development, all discussions ultimately rest on tacit understanding, referring to the intended application context. For a formal document to be meaningful, we have to ensure, by suitable informal procedures such as the careful reconstruction of concepts, as used in the relevant application area, or by arguments giving the context and the decisions that have led to the specific formal model, that it relates to our tacit understanding.

Formal methods

A method offers guidelines, consisting of techniques, tools and forms of organization. In a formal method, the techniques and tools centre around the use of a formalism, for example a language with a formal syntax and formal semantics. By contrast, languages (and methods using them) are called semi-formal if they have a formal syntax but no formal semantics. People also speak about informal methods, but, in my opinion, this term is only fit for colloquial use, since it indicates the absence of formalism but not the presence of anything else.

I see important differences between formal and semi-formal languages in their use as communication media between people (for example, when writing specifications):

- Semi-formal languages can be used for partial, incomplete descriptions arranged in a formalized manner, whereas whatever is described with the help of formal semantics must be complete and unambiguous; this is normally required only for texts which are to be machine-processed.

- Semi-formal languages do not provide the basis for formal correctness proofs, claimed to be one main advantage associated with the use of formal methods.

In discussing the relevance of formal methods to software development, we have to distinguish between methods on two levels: global methods addressing software development as a whole, and components of such methods supporting specific tasks with local applicability. Formal methods can only appear as component methods, to be combined with other component methods offering semi-formal languages and techniques for writing prose-texts, as well as informal methods to be discussed later. For formal methods to be useful, their combination with other method components and their embedment in software development as a whole must be carefully designed since, surely, it is the process as a whole, and not the local use of any particular method, that determines the quality of the resulting product.

The formalist argument and its critique

As far as I am aware, formalists, in advocating their methods, rely on the following view of software development:

Software development starts from fixed requirements, which are given but, as yet, not usually formally expressed (it is to be hoped, from the formalists' point of view, that suitable formal techniques will soon be available for handling requirements). The task of the software developer is to transform these fixed requirements into a correct (and efficient) program. This transformation occurs in several steps. First, we need to specify WHAT the program does in abstract, mathematical terms without regard to HOW the program is implemented. Then, for any given programming language, the corresponding program can be derived in one or more steps which themselves can, and should, be formalized. As opposed to conventional programming, the resulting program can be proved correct with formal proof techniques. If needed, it can subsequently be transformed into a more efficient solution.

In the view of formalists, the use of formal methods leads to important gains:

1. The process of formalization (abstraction) in the early stages of software development will promote the finding of errors, inconsistencies and missing elements in the requirements.
2. On the basis of the specification, software developers will be able to answer questions on the intended functionality of the program at a very early stage.
3. Programs can be proved correct with respect to their specifications.
4. Specifications can be transformed into programs according to fixed rules ("mechanically").
5. In some methods, specifications can be executed and thus serve as a prototype for the future system.

In giving a critique of the formalist position, I will disregard questions of training, motivation, resources and tools. I will confine myself to the questions: To what extent is the formalist standpoint relevant to software development? Are its assumptions valid? Are its claims justified?

In doing so, I will draw on a different, process-oriented view of software development, which can be summarized as follows:

Software development consists of processes of learning, creative cooperation and communication involving people of differing backgrounds and interests. Its object is not so much a product derived from fixed requirements, but a change in human work processes involving the use of this product. While there is a class of well-defined functional requirements, which may be determined in advance and perhaps even formalized, this does not hold for those aspects that determine the suitability of a program as a tool for human users: the distribution of functions between man and machine, the matching of system functions to work-steps, the possibility of human intervention in the case of system errors, and the handling of errors in the context of human work. These aspects are not amenable to formalization since they cannot be described completely and unambiguously in advance. Furthermore, software requirements change with the needs of the user organization.

Therefore, software development must be understood in cycles of (re-)design, (re-)implementation and (re-)evaluation in which processes of software development leading to system versions are interleaved with processes of system use; since software quality is determined in using software, the development and use of software must be considered together.

In this view, formal methods fail to address key issues in software development:

- finding out what are the relevant requirements and how they relate to each other;
- designing system functions and software architecture taking into account both the informal context of the system's use and the specific properties of the underlying base machine;
- making software systems suitable as tools for people;
- adapting existing systems to meet new needs.

Against this background, several benefits of formal methods, as claimed by their proponents, appear doubtful:

- ad 1. Formalization is only one activity that will lead to finding errors and

missing elements, and it is not necessarily the most reliable one. In particular, it does not point to discrepancies between the requirements as formulated and the real needs of the application context.

- ad 2. "Understanding" system functions on the basis of a formal specification means understanding their abstract mathematical properties; it does not enable us to answer reliably questions pertaining to the use of these functions in human work or problem-solving.

- ad 3. Proofs essentially rest on argumentation; they are established, found convincing or revised in a social process involving learning and constructive criticism; formal proofs are subject to the same errors as programs and would themselves have to be proved by informal argumentation; therefore, it is not obvious what is gained by formal correctness proofs of programs; moreover, it is not clear what is gained by a "correct" program since the specification rests on shaky ground, being the formalized version of the result of an unreliable communication process early in software development.

- ad 4. I know of no case where a programmer has derived a program mechanically from a specification, and I think there are good reasons why programmers proficient in formal methods, after having written the specification, have to make a design which takes the specification into account, but cannot be derived from the specification by any fixed rules. There are large parts of programs which are beyond the scope of formal specifications: details of man-machine interaction, formats of input and output data, communication mechanisms with peripheral devices and software utilities; they also obey different quality criteria, such as efficiency and storage economy; a new design will, therefore, lead to better results.

- ad 5. A specification that can be executed by a computer is a program; it may well be that improved programming languages based on formal semantics will constitute an improvement on existing languages, but this leaves open the problem of specifications that are useful for humans; such a specification cannot be a formal document only, and it must be geared to the actual needs in communication between the people involved, rather than to the prerequisites of mathematical theories.

Towards a balance between formal and informal methods

The purpose of this conference is not, of course, to confront two irreconcilable positions, but to work towards a harmonious interplay between process and form at all levels in software development. Therefore, the two standpoints sketched out above need to be seen as complementary rather than contradictory.

Even in the process-oriented view, we work at any point in time towards a product whose requirements are considered fixed. On the other hand, every formalist will readily agree that formal methods are applied in a context of changing human needs. Furthermore, while both views can be applied to all software systems, their relative importance depends on the type of system to be constructed and how it is embedded in its usage context. For example, compilers for a given programming language have a well-defined desired functionality and are therefore vastly more amenable to formal approaches than interactive application systems where the interleaving of computer-supported and other work-steps of predominantly lay users is a major concern. Therefore, not all points of the critique of the formal argument carry equal weight in all situations.

As a sceptic with respect to formal methods, I can nevertheless see their usefulness in several important respects:

- to clarify concepts underlying our software work, as has happened in the case of abstract data types and modules, which have emerged, as a result of work in formal semantics, as vehicles for focussing our thoughts and discussions in software design and which can be studied and practised on the basis of small examples; personally, I find this help immensely valuable, and I believe that a large portion of the positive experiences reported by users of formal methods is due to this effect;
- for literal application in well-defined sub-problems, where concerns of functionality and reliability take precedence over concerns of usability and where requirements can be expected to remain stable; in my view, the treatment of concurrency falls into that category;
- to serve as an inspiration or guidance for organizations developing their own methods; I feel that the emphasis here must be on retaining the spirit of the method while deliberately omitting the literal use of some of its more formal elements and adding suitable informal elements geared to supporting processes of learning and creative cooperation;
- to standardize solutions in problem classes which are already well understood and where the problem of designing new systems in a creative process can, to some extent, be replaced by the use of known solutions;
- to provide new ways of programming with the help of languages based on formal semantics, which may well avoid the serious shortcomings of existing programming languages while at the same time permitting the formulation of programs at a higher level.

On the other hand, I feel strongly that processes of learning, communication and creative cooperation cannot, and should not, be formalized, although attempts at doing so are wide-spread amongst authors of software design methods, for example when structuring criteria like "top-down" are applied to the design process as it takes place in time, rather than to the form of its result. In my own experience as a software designer and project adviser, processes cannot be entirely formalized, and attempts at doing so prove positively harmful.

Such attempts to reduce process to form are based on a serious misunderstanding of the nature of processes. Processes are irreversible; they take place in time. By their very nature, they cannot be fully understood and described in advance; they can only be understood by the people involved in these processes as they happen. Here, people act in open situations, on the basis of their needs and commitments, pursuing changing goals as they go along and revising their understanding in the event of new insights. If we wish to understand the nature of such processes, we have to develop a new theory drawing largely on concepts that are useful for studying living systems, as they exist in nature and society, and language as it is used in communication between people.

Though no such theory is available at present, we can nevertheless suggest concrete informal methods to support processes of communication, which are in wide use:

- working out relevant examples, taking into account the application context
- taking care in the choice of basic concepts used in software development so as to enhance common understanding
- constructive criticism of partial solutions or partially formalized solutions on the basis of transparent criteria
- carrying out controlled experiments, for example in the form of prototypes, which can be evaluated
- using psychological methods for improving the communication between groups of people enacting different roles.

Whether or not formal methods are applied, these informal procedures may be expected to bring enormous benefits.

I should like to close this introduction with a few personal words. My involvement in the TAPSOFT conference must be understood in terms of my friendship with my colleague Hartmut Ehrig, which is based on great mutual respect at a personal level, while at the same time we most heartily disagree: he is a powerful advocate of formal methods, whereas I remain an undaunted sceptic in this respect. However, because of

our friendship, it has not been possible for us to dismiss each other's viewpoints as irrelevant; so, instead, we have carried on a continual heated discussion over several years. As a result, I now have a fairly good appreciation of what algebraic specification can or cannot do for me, under what circumstances and for what purposes I would use it in software development, and how it could be embedded in the overall approach which I favour. I am sure, too, that Hartmut has a better appreciation of the importance of those aspects of software development which cannot be formalized. Yet, I continue to work without feeling any compulsion to become proficient in algebraic specification, while he continues to "abstract from" those unpleasant aspects of software development which are not amenable to formal approaches, in order to make progress in elaborating his specification technique. Thus, our disagreement is not resolved, and it cannot be, since it rests on both sides on differing fundamental convictions. And yet, as we share a common concern for greater quality in software development, the basic question in my mind is how these two viewpoints can fruitfully interact and be combined to produce workable approaches.

I have borrowed the phrase "a balance between formal and informal methods" from Don Knuth. In his enlightening paper on "Literate Programming" he writes: "I owe a great debt ..... to Peter Naur for stressing the importance of a balance between formal and informal methods". I owe Peter Naur a much greater debt in this respect, since he has for years been my main support in developing my own ideas in a context where contrary views were for a long time the predominant ones.