Formal Aspects of Computing (1997) 3: 1–000

Limits of Formal Methods

Ralf Kneuper
Philipp-Röth-Weg 14, 64295 Darmstadt, Germany

Keywords: Formal methods; software quality; correctness; trustworthiness; formal specification

Abstract. Formal methods can help to increase the correctness and trustworthiness of the software developed. However, they do not solve all the problems of software development. This paper analyses some limitations of formal methods.

1. Introduction

When discussing the possibilities and limitations of formal methods, some people (often the 'academics') take either a highly optimistic view, stressing possibilities and ignoring limitations, or (often developers from 'the real world') a highly pessimistic view, describing certain limitations of formal methods and deducing that, since formal methods do not solve *all* our problems, they are useless. However, during the last few years, more people have started to promote a realistic view of the applicability of formal methods (e.g. in [CDH+96]).

The main goal of this paper is to support this realistic view of the possibilities and limitations of formal methods (concentrating, as the title suggests, on their limitations, because the readership of this journal will already know all (well, almost) about their possibilities) To some extent, these limitations will seem quite obvious once stated, but in the author's experience, some parts of the formal methods community still do not fully realize them.

Such personal experience led the author to write the current paper. It stems from working in a formal methods research group for several years (cf. [JJLM91, Kne91], where he developed a rather optimistic view concerning the applicability of formal methods. Working first in the quality management group of a major software house and now in the IT department of a large company, he soon had to

Correspondence and offprint requests to: Ralf Kneuper, Philipp-Röth-Weg 14, 64295 Darmstadt, Germany

appreciate that there is much more to producing 'good' software products than using formal methods. This experience led to the current paper.

1.1. Formal methods

There are two fundamental views of formal methods as a discipline:

- as a branch of pure mathematics, an intellectually challenging research field which may or may not have any application in the "real world", or
- as a branch of software engineering which is concerned with the design and application a certain set of development techniques and tools to create better software systems

Both of these views are legitimate and useful. Problems arise if, for funding and similar reasons, the first view hides behind the second. This paper will be based on the second view since the limitations discussed refer to the practical applicability of formal methods (and, admittedly, this is what the author is most interested in today).

> Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates. [Fai85, p. 2]

As for the term formal methods, we use the following definition:

> ... a formal method is a set of tools and notations (with a formal semantics) used to specify unambigously the requirements of a computer system that supports the proof of properties of that specification and proofs of correctness of an eventual implementation with respect to that specification. [HB95b]

Typical techniques used in formal methods are invariants, proof obligations, and a calculus for refining specifications or proving properties about specifications and implementations, and the relationship between a specification and its implementation.

The emphasis here will be on formal methods concerned with the *functional* requirements and correctness, e.g. VDM or Z, but there are also formal methods concerned with other quality characteristics, such as for dealing with performance requirements.

Formal methods can be applied at different levels, ranging from 'only' writing formal specification for small parts of a system, via the *rigorous* approach of expressing but not usually discharging proof obligations, to providing formal proofs of program correctness with respect to the specification, or even proving that the compiler, its environment and the hardware satisfy their specifications (under stated assumptions). Formal specifications are usually considered the most important and most useful aspect of formal methods:

> From an economic point of view, therefore, the most important part of a formal development is the *system specification*. For many projects, this is the only part of the development that is formal. [Hal90, p. 13]

1.2. Goals of using formal methods

As a branch of software engineering, formal methods are concerned with the systematic production and maintenance of software products, on time and within

cost estimates, using a certain kind of tools and techniques. Specifically, they are concerned with correctness and high reliability of the resulting software system, mainly achieved by

1. supporting the creation of specifications that describe the true requirements of the user, which are not usually identical to the requirements stated. However, whether formal methods can help with this task is a matter of contention. While proponents of formal methods claim that this can indeed be achieved using formal methods because of the unambiguity of formal specifications and the possibility to prove certain properties about it, opponents state that on the contrary a formal specification is incomprehensible to the average user and is therefore even less likely to be correct (cf. 3.2).
2. ensuring that the implementation satisfies the specification
3. increasing trustworthiness in the sense that the system developed is not just correct but *known to be correct*. When there are high demands on reliability and correctness, evidence is needed that a given system indeed satisfies these demands. A system for which no such evidence is available will not be acceptable no matter whether it is indeed correct or not.

1.3. Structure of this paper

Section 2 covers a number of issues not addressed by formal methods but necessary for developing high-quality software. Section 3 describes the limits of formalization in general, which might also be called theoretical limits of formal methods.

Practical limits of formal methods are dealt with in Section 4. Depending on the degree of formality, a complete and formal description of the environment of the software developed (hardware, operating system, compiler, etc.) is needed but rarely available. Once the technical problems of using formal methods are solved, one needs to transfer the technology into practical use. This involves convincing and training developers as well as providing adequate tool support.

In Section 5, the pragmatic limits of formal methods are desribed. These do not prevent their use but are the reason why formal methods are not always useful and sometimes not an efficient way to achieve the goals stated.

The paper ends with the conclusions in Section 6.

2. Issues not addressed by formal methods

2.1. Software product quality

Formal methods deal with the software itself and, to some extent, its documentation. Other important components of software products such as training, customer support or installation of the software, have to be dealt with separately.

Together, these components and their quality form a considerable proportion of the quality of software products, and can make or break a product just as much as the correctness of the software. As a result, successful providers of software products put a lot of effort into addressing *all* relevant aspects of a software product.

This is often done by introducing a *quality system*, as for example described

in the ISO 9000 series (for software development see in particular ISO 9000 Part 3 [ISO91]). The idea behind such a quality system is to consider software development as a process consisting of many steps, each of which influences the quality of the final product. In order to predictably get a quality product at adequate cost, one needs to assure the quality of the complete process, including e.g. contract reviews, configuration management, and corrective action in case of problems. Only if each individual component of the development process is dealt with adequately, one can expect to deliver a quality product. Specification, design and development methods (including formal methods) form only part of a quality system for software development.

The same basic idea is used in the SEI (Software Engineering Institute) capability maturity model [Hum88]. This model defines five different levels of process maturity that an organization developing software can reach, as measured in areas such as requirements management, configuration management, project planning, training program, process measurement, etc, and suggests priorities for improving the development process. Organisations reaching level 3 in this model (characterized as *defined*) are considered as having full control over their development process.

To be useful, the usage of formal methods must be embedded in such a quality system covering the full development process, to ensure that the advantages of using formal methods are not lost due to simple mistakes in other areas of the development process.

Furthermore, formal methods only address certain aspects of software quality, mainly correctness and reliability. Other aspects, such as efficiency or user friendliness, are addressed only to a minor extent, if at all. However, although correctness and reliability are important quality characteristics, they certainly are not the only ones. It is sometimes claimed that without correctness, other aspects of quality such as efficiency are irrelevant since one cannot rely on the result. Although there is some truth in such a claim, it is certainly not the whole truth, as can be seen from the fact that very few, if any, of the programs used today are fully correct. Nevertheless, many of them are useful.

Similarly, formal methods give high priority to abstract properties of the output (or the input-output relationship, to be precise), but ignore issues of human-computer interface, such as number representation, line and page format, or output medium (cf. [Nau82, p.441]). There are few attempts at formally describing human-computer interface issues, which cover mainly dialogue structures (e.g. [Mar90]).

2.2. Software systems and their social and ecological environment

A seemingly obvious but often ignored issue is the fact that software systems do not stand on their own but are embedded in a social and ecological environment consisting of different types of users as well as society and the natural environment around them. This affects the problem of developing 'correct' specifications and deciding what behaviour is correct. Here formal methods can contribute nothing and do not attempt to do so.[1]

[1] To be honest, this is not just a problem of formal methods but of software engineering in general.

Typical issues that must be dealt with are the effects of a software system on the working conditions of those working with it, their training and their acceptance of the system.

Looking outside the user organisation, issues to be considered include e.g. data protection. Occasionally software systems become a problem because they work too well (e.g. computer trading at the stock exchange has in some instances led to considerably higher price volatility).

In [CNP+92, p.14], Rolf and Siefkes describe this as follows (Translation by the author):

> Computers are "symbolic machines" ..., programs are formal instructions for computers; therefore, mathematics is the proper science for them. But not the only one. The use of computers affects not computers but people, often also nature. So why this modesty? Computer scientists do not construct computer systems, they manipulate social and ecological systems by constructing computers into them.

3. Limits of formalization

3.1. Complete formality

As described above, there are different degrees of formality in developing software, but complete formality is impossible to achieve. Naur states these limits of formalization as follows:

> "What will be argued here is that in reality the meaning of any expression in formal mode depends entirely on a context which can only be described informally, the meaning of the formal mode having been introduced by means of informal statements."[Nau82, p.439]

Complete formality, though it might seem desirable, is not possible, because the underlying basis, mathematics, can itself not be formalized completely. An informal starting point to build on is always needed, such as the meaning of the symbols of an axiom in a theory.

Standard mathematics is actually not very formal at all in the sense that little of the language used for expressing mathematical statements is formally defined, most reasoning takes place in a natural language interspersed with some formal expressions. Proofs in particular are rarely broken down to the level of explicit appeal to the axioms, lemmas or theorems used.

Naur further argues that formal expressions are only used as abbreviations when informal expressions would be too long and cumbersome to use. However, they are not really necessary, all that can be said "in formal mode" could also be said "in informal mode" — but not vice versa.

Although correct, this argument misses the important point that formal expressions can be a huge help when expressing or reasoning about complex issues. Indeed, due to limitations of the human brain, many insights are not possible without adequate formal notation, even though once found they can be expressed in an informal way as well. For example, inconsistencies in a specification are easier to find and misunderstandings less likely to happen if the specification is presented in a well-structured form, which is supported by the use of a formal language.

3.2. Correctness of specifications

Even when using formal methods, there is no way to guarantee correctness and completeness of a specification with respect to the user's informal requirements. There are various approaches to reduce the probability of incorrect specifications (the author's work described in [JJLM91, Ch. 9 and App. D] was one of them), but since the starting point is necessarily informal, one can never be sure to have gathered all user requirements correctly. Additionally, true user requirements might be different from what the users say they need, and will usually vary with time.

This is, however, a fundamental problem of *all* software development methods, independent of the degree of (in-) formality, and it is one of the goals of formal methods to reduce it as far as possible. Whether they succeed more than other development methods is a question that cannot be answered in general, since the answer will depend on factors such as the specific method and tools used, the structure of both the problem domain and the specification used to describe it, and the mathematical maturity of the users who check the stated requirements. Formal methods have the potential to achieve more than informal methods with respect to correctness of the specification, since they provide a unique meaning for the stated requirements and reduce (ideally prevent) ambiguity. They offer a basis for reasoning about the specification and a number of heuristics for checking the completeness and correctness of a specification (such as the *proof obligations* for VDM [Jon90]).

The formal methods used today do not fully achieve this potential, since the notations used are very difficult to understand for people not fully familiar with mathematical notations, such as typical users of software. As a means of communication with the customer, a notation such as VDM or Z is not suitable. Hall mentions three ways to make formal specification comprehensible to the user:

- Paraphrase the specification in natural language.
- Demonstrate consequences of the specification.
- Animate the specification.

> The first way is always essential. A mathematical specification must be accompanied by a natural-language description that explains what the specification means in real-world terms and why the specification says what it does. [Hal90, p.18]

A promising approach to simplify the task of paraprasing a formal specification in natural language is the GIST paraphraser (described in [Swa82]), which translates a formal specification (written in GIST) into natural language. While an automatic translation from natural to formal language is, in general, impossible, the reverse translation is not too difficult and can be very useful for making (formal) specifications easier to understand. Actually, the experience of the developers of the GIST paraphraser showed that the different view provided by such a paraphraser even helped them to discover problems in a specification that they had not noticed in the formal specification itself, in spite of their experience with formal notation. On the other hand, had they started writing an informal specification, it is unlikely that this specification would have turned out as well-structured as the informal specication derived from the formal one.

An alternative approach is to use common graphical techniques such as data flow diagrams or entity relationship diagrams and provide them with formally defined semantics so that they can form the basis of a formal development ap-

proach. The paper by Larsen, Plat and Toetenel [LPT94] is an example of this approach and provides many references to similar work.

3.3. Correctness of implementation

In general, it is undecidable whether or not a given program satisfies a given specification, i.e. whether an implementation is correct. For example, when using a verification approach such as Hoare logic, one needs to identify the loop invariants, which is not possible automatically.

As a result, it is very difficult and often impossible to prove the correctness of an existing program that has not been written with the correctness proof in mind. Correctness proofs are only feasible if programming and proof go hand in hand. If programmers think about how to prove the correctness of a program while they write it, they will write a different program. They will be able to prove the program correct because when they write a program statement they (hopefully) know why it achieves what is needed.

> ...you construct a correct program in small steps. Each step takes the specification and produces something a little nearer to the to the final program. Each step is small enough that you can see exactly what needs to be proved to show that the ste is correct — and, if in doubt the correctness, you can actually carry out the proof. [Hal90, p. 15f]

This approach is close to trying to ensure correctness by transformations (e.g. [PM87] rather than creating a proof. If the transformations are automated, then this is equivalent to programming at a higher level of abstraction, sometimes also called executable specifications.[2]

There are three possible reasons why the proof of correctness of an implementation with respect to its specification might fail:

- The program is actually incorrect and needs to be modified
- The program is correct, the correctness proof just has not been found yet
- The program is correct, but there is no correctness proof, for example because the proof depends on an unprovable statement or, put differently, the proof system used is too weak. Although this case may, in theory, occur, it implies that the programmer does not and cannot know whether the program written is correct. This case therefore should never occur in practice, or if it does, the program should be 'corrected' just like an incorrect program, or, possibly, the proof system needs to be modified.

The difficulty here is that it is not decidable which of these three possible reasons applies. So if after serious searching no proof has been found, there is no real alternative to revising the program based on a new analysis of how it is to implement the desired functionality.

Of course, a proof of correctness is not necessary for a program to be correct. However, in many cases the fact that a program is correct is of limited use as long as we do not *know* this, or at least have good reason to believe it. The difference here is similar to that between truth and provability of a statement. Put differently, the program needs to be *trustworthy* as well as correct.

---

[2] Calling this "specification" rater than "programming" is comparable to the introduction of third-generation languages such as Fortran and Cobol. At the time, this was considered as "automated programming".

One of the curious things about the trustworthiness of a program is that it does not really depend on the correctness of the program itself (which, in general, is not known) but on the correctness of other programs developed under similar circumstances and using similar methods (which is assumed to be known). Even if a program has been proven to be correct (under appropriate assumption), the proof itself might still be faulty, and we therefore only consider the program trustworthy because of previous (hopefully positive) experience with similar proofs.

Additionally, the compiler is itself a software system and as such unlikely to be fully correct. As a result, if very high trustworthiness of the system to be developed is required, formal methods must have been used in developing the compiler as well as the system itself. On the other hand, if such a compiler is not available, this does not restrict the *use* of a formal method but limits the confidence in the results.

3.4. Correctness of proofs

There are two main reasons why correctness proofs (and, as a result, the correctness of proofs) play an important part in formal methods. Given that it is usually impossible to know for sure that a program is correct, correctness proofs at least increase the probability that the program is correct and thus increase the trustworthiness of the program. This applies to complete correctness proofs as well as to proofs of individual properties of a program, its specification or the relationship between a program and its specification.

As described by De Millo et. al., 'mathematical' proofs are checked in social (and therefore fallible) processes before they are accepted as correct.[3]

> We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem — and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs. [MLP79, p.271]

In verification of programs, such a check is only possible to a very limited extent using reviews, inspections, etc.

One approach to increase confidence in the correctness of a program and its proof is to automate the correctness proof using some form of theorem proving tool. Unfortunately, we are dealing with a problem here that is not computable, and even restricting oneself to a reasonably large class of special cases, such an automatic program prover is, in spite of a lot of research in this area, not currently feasible.

Additionally, even an automatically generated proof can still contain errors, although this would hopefully be considerably less likely than for a manually generated proof.

A *proof checker* (as opposed to a theorem *prover*) would be much easier to build (for a start, we are now dealing with a decidable problem) and provide a very high degree of confidence in the correctness of a program "proven" correct. Such a tool only takes given proofs and checks them for correctness, without trying to generate any proofs itself.

---

[3] A good example of this process is the recent discussion whether the claimed proof of Fermat's Last Theorem by the mathematician Wiles is valid.

The main problem here remains in creating the proofs in the first place. In particular, to check proofs automatically for correctness, the proofs must be *formal* proofs rather than proofs in the usual mathematical, semi-formal style — which implies that, in practice, they are close to impossible to verify by hand using the social process described above.

3.5. Abstract machine and target machine

As noted by Fetzer [Fet88, p. 1058], there is a difference between the correctness of a program (on an abstract machine) and the correctness of any execution of the same program on any physical target machine. A layer consisting of the compiler, hardware, and various other parts of the environment separates the two.

To model this layer, a formal description of the environment that the program is to work in is needed. Such a formal description is a mathematical model of certain aspects of "the real world". As is always the case for such models, there is no guarantee that they describe all important features of our physical environment. This means that e.g. after a hardware failure, the model is no longer correct and a program may no longer work as intended even though it was proven correct.

Therefore, programs that are run on an actual computer can only be verified relative to the environment provided by this computer, while absolute verification is only possible for running the program on an abstract machine but this is not usually what users are interested in. From this, Fetzer deduces

> The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility. [Fet88, p.1048]

Put like this, the statement is of course true but not surprising since the same argument holds for any method that is used for making statements about the physical world.

As a result of the possibly different behaviour between abstract machine and target machine, the traditional approach of testing software in order to validate and verify it cannot be replaced by formal methods, even though testing, too, is quite insufficient on its own. The two approaches can be combined, using a formal specification to generate test cases and check that the test results are correct, i.e. satisfy the formal specification [HP95].

4. Practical limits

Most of the limits discusssed so far concerned the effectiveness of formal methods for improving correctness and trustworthiness of software products. We now consider some limitations that make formal methods, and in particular correctness proofs, difficult to apply.

4.1. Dealing with complex language features

For many important language constructs and software system components, formal definitions of their semantics are either not available or too complex to be

useful. However, in order to prove properties of programs using these constructs or components, such a describption would be necessary. Examples are

- complex data structures
- pointers
- human-computer interface (HCI) and error messages. DeMillo et. al. mention estimates that more than half the code of any real production system consists of HCI and error messages [MLP79, p.277]. Of course, these can also in theory be formally specified and, if considered necessary, the implementation verified, but again the resulting expressions get very complex and cumbersome to handle. It seems very unlikely that this would help to achieve the main goal of the HCI, making the system developed easy to use.

Rounding errors and size limitations could also be included in this list but will be dealt with as part of the technical environment described in Section 4.2.

For all these limitations one can argue that the actual problem is not the complexity of the techniques needed to deal with them in a formal manner, but the complexity of the language features themselves. Formal methods just make this existing complexity visible and force the developer to deal with it explicitly. For example, pointers are not just difficult to deal with using formal methods, but are also a common source of errors in programming in general.

4.2. The technical environment

In order to prove the correctness of a program, a formal description of the environment a program is to work in (e.g. hardware and operating system) must be available, in addition to the formal definition of the programming language and its features. Such a formal description is often not available for the kind of technical environment used in industrial software development even though, in principle, it is quite possible to create it. The problem is made worse by the fact that such a formal description has to take a very specific form depending on the formal method used (for example as a theory to be used in a theorem prover).

This applies to both the development environment and the production environment. As for the development environment, a formal definition of the programming language used and its semantics *as implemented in the compiler* is needed. For languages other than toy languages this brings us back to the problems with complex language features described above.

Once development of a system is complete, it is (hopefully) put into production in a production environment. This usually consists of hardware, operating system, and many other systems such as TP monitor, DBMS and drivers for various peripherals such as printers. For each of these, the exact semantics of all interaction with the system must be defined.

Additional complications are introduced by the following aspects of the environment:

- rounding errors in computations with floating point numbers. These are the reason why formal methods are not usually applied to numerical algorithms.
- size limitations. These can be handled ("clean termination" [CH79]) but the resulting specifications get very complex. For example, the fact that only a finite number of integers can be represented on a computer can be described by introducing upper and lower bounds on integers. In that case, however,

standard properties such as the associativity of addition no longer hold in general.

The relevance of these limits depends upon the level of formality applied. As far as formal specifications are concerned, they do not really make any difference. However, as soon as one starts to prove any properties of the implementation, one needs to formalize some aspects of the technical environment. However, most of these aspects are usually ignored in order to concentrate on the main functionality of the system to be developed, accepting that e.g. size limitations need to be dealt with in some other way.

4.3. Scalability of formal methods

Most current formal methods are mainly applicable to small-scale applications, but do not scale up well.

> The classic formal methods fall into the small-grain category. These methods have a mathematical basis at the level of individual statements and small programs, but rapidly hit a complexity barrier when programs get large. In particular, systems for reasoning with pre- and postconditions — such as Hoare axioms, weakest preconditions, predicate transformers, and transformational programming — all have small-size atomic units and fail to scale up because they do not provide structuring or encapsulation. [LG97, p.79]

However, as the examples suggest, scalability of formal methods is mainly a problem when trying to prove correctness of an implementation with respect to its specification. As far as the use of formal specifications is concerned, scalability is less of a problem since, even for large systems, the formal specification need not always be very large due to the high level of abstraction possible with a formal specification language. Furthermore, it is not always necessary to specify all of the system formally.

There are a number of approaches to scale up formal methods, see e.g. [LG97, p.79–81]. Nevertheless, a lot of work remains to be done before they are really applicable to large-scale applications.

4.4. Formal methods and the developers using them

Software development is done by people, not by machines. No matter how 'good' a development method is, it will only be successful if the developers who are to use it are willing and able to do so.

In the case of formal methods, it is sometimes claimed that, in order to be able to use them, you need to have a Ph.D. in mathematics. Although this is overstating the problem, it is true that the use of formal methods requires a higher degree of mathematical maturity than the average software developer possesses today.[4] Obviously, these requirements vary with the specific task to be done. Reading and implementing a formal specification is much easier than writing it in the first place, while a correctness proof is much more difficult still.

Some proponents of formal methods therefore argue that there should be higher entry requirements for software developers, just like an engineer needs to

[4] Admittedly, there are different opinions on this claim. E.g. "*Certainly, anyone who can learn a programming language can learn a specification notation like Z*". [Hal90, p.16]

be able to master a large amount of mathematics before being allowed to design a bridge, a house or a car.

On the other hand, this would not just amount to introducing a new development method but to restructuring the profession of software developer, a task that should not be undertaken lightly, assuming that it is possible at all. For example, this could eliminate many developers who think along different lines and make a valuable contribution by coming up with new product ideas, or communicating with users and colleagues. Communication forms a considerable proportion of development work. According to [BSH+93], developers spend on average about 31% of their time on activities requiring communication skills (meetings, presentations, discussions, etc) while technical activities (specification, coding, testing, etc) take up about 55% of the time.

So far, mainly the *ability* of developers to use formal methods has been discussed. Even more important is their *willingness* to do so. There is a lot of opposition among developers to even semi-formal approaches to software development, let alone formal methods (this is of course partly due to their inability to use them).

Introducing formal methods in an organisation poses not only technical but also many social and psychological questions. Therefore, such 'technology transfer' usually takes a lot of time and one should not expect to introduce formal methods quickly in most environments.

4.5. Tool support

Tools are available for most of the tasks to be done when using formal methods, on different levels of formality. Examples are syntax-directed or graphical editors for specification languages, theorem proving tools (automatic or interactive theorem provers and proof checkers), code generators, interpreters for prototyping of formal specifications, tools for (semi-) automatic test case generation and test evaluation, and compilers for programming languages with assertions. Good tools can reduce some of the problems raised above. For example, dealing with size limitations is not actually difficult but highly cumbersome and involves a lot of clerical work most of which could be automated.

The following problems currently hinder the development and usage of tools.

First, in order to provide genuine support for formal methods, standardized languages and methods are needed. In most cases, these are so far not available. For example, there are several different dialects of the VDM specification language and as a result, different tools supporting VDM can rarely be combined.[5] Most current tools are quite inflexible in the sense that they each support one specific language (dialect), method, logic, etc. This makes their combination very difficult.

The *mural* interactive theorem prover [JJLM91] showed one approach to solving this problem since in this system very little of the underlying logic and theories is built in. The three-valued Logic of Partial Functions LPF used in VDM is provided as an example instantiation, but the user can define other logics and theories instead (with few restrictions, e.g. monotonicity).

[5] However, a draft international standard (DIS) has been produced by an ISO working group for the VDM specification language VDM-SL. Another group is working on a similar standard for Z.

A second problem in using tools to support some tasks in using formal methods is the need for considerably more formalisation (see Section 4.2). Few tools support a mixture of formal and informal work, and it is difficult to define what kind of support one would want.

The small market for such tools causes an additional problem. As long as few tools supporting formal methods can be sold, few will be developed, and the ones that are developed are mostly developed in an academic environment where there are few resources and little motivation to turn a prototype into a real product with support and maintenance etc.

(Missing or inadequate) tool support is sometimes described as one of the reasons why formal methods are little used. Although of course better tool support is likely to increase usage of formal methods, the effect is likely to be small. Good tools can reduce the effort for clerical and routine work considerably, such as is needed in proving programs correct. However, before being able to automate a development method (or, generally speaking, any kind of work), one first needs a good understanding of the method itself ("*a fool with a tool is still a fool*").

5. Limits to the usefulness of formal methods

There are a number of environments where one could use formal methods but it would not be useful to do so, mainly because costs are too high in relation to the benefits gained. In contrast to the limitations discussed in Section 4, all the necessary tools etc may be available but it still does not make economic sense to use formal methods.

The most important of these "pragmatic limitations" on using formal methods is closely related to the fact that formal methods only address a limited range of issues in software development (cf. Section 2). Doing some parts of software development really well using formal methods is of little use if important other tasks are hardly done at all. E.g. it is wasted effort to prove a program correct (under stated assumptions) if one cannot be reasonably sure about sending the correct version of it to the customer, i.e. if there is inadequate configuration management/version control. Essentially, this implies that it is only worth putting the effort into applying formal methods if the full development cycle is well under control, for example in the sense of having a quality system according to ISO 9001 or having an organization on level 3 of the SEI Capability Maturity Model. Since currently the majority of organisations developing software still have a long way to go to achieve this goal, introducing formal methods would be wasted effort for them.

Another fairly obvious pragmatic limitation is that since formal methods mainly help to increase correctness and reliability, they are only useful if there are fairly high demands on the correctness and reliability of the resulting system. Although this will often be the case, there are a number of software systems where other quality characteristics have much higher priority. An extreme example are many games where the main priorities are a good graphical user interface and speed of the game software. For such a system, formal methods are of little use.

A type of software that tends to be difficult to handle using formal methods are commercial applications that often contain hardly any algorithm. Their complexity stems from the large number of input and output fields but the 'algorithm' consists mainly of moving data around. This makes them difficult to

specify formally, a formal specification could easily be just as long and as tedious as the implementation itself.

Additionally, the requirements for commercial software often change very fast, partly because they are not well-defined from the outset and partly because its environment changes very fast. Although this can make formal methods more useful by helping to keep track of the changing requirements, it also increases the effort needed considerably. This may be acceptable when writing formal specifications of the system, but makes it unacceptable to prove programs correct each time unless the requirements on correctness and reliability are very high indeed.

Of course, this argument not only applies to commercial software but to any kind of software with rapidly changing requirements.

So far, we have mainly dealt with the developers' point of view and the reasons why they do not use formal methods. Just as important are the points of view of management and of the customers. The reasons why they decide for or against a certain development approach are mainly based on economic rather than technical criteria, such as the perceived cost-benefit ratio and the risk involved. Here convincing case studies are needed that look at the costs as well as benefits, and express them both in financial terms. Looking for example at the case studies in [HB95a], some (such as [FLBG95]) cover the costs of formal methods as well, while many others only look at the results of the case studies in technical terms.

6. Conclusions

Even using formal methods, writing formal specifications and correctness proofs can only help to increase the likelihood but provide no absolute guarantee that the resulting program is correct. As Brooks put it, there is no "silver bullet" to solve the problems of software engineering, and this includes formal methods.

The main reasons for this are

- the fact that formal methods cover only parts of the development process
- the possibility of creating an incorrect specification which, even when implemented correctly, still results in a faulty program
- the possibility that, in spite of applying formal methods, one can still create an incorrect implementation of a specification, either because the description of the relevant environment (TP monitor, complex programming language features such as pointers, etc) is incomplete or incorrect, or because of plain human error in applying formal methods.

When should formal methods be used? Obviously, in different environments, different levels of formality will be useful.

Formal specification is often useful in order to increase completeness and consistency and reduce ambiguity. Unclear or ambigous requirements are a common source of defects of software systems, and the effort to use formal specifications is not too big. However, whether or not formal methods help to solve this problem also depends on the kind of software to be developed and the sophistication of the users who are to judge the correctness of the specification. Commercial systems with little algorithm are less suitable, while for software with complex algorithms, formal methods can be of great help.

Other typical situations where it is useful to create formal specifications are the definition of interfaces or of standards (e.g. of a programming language and its semantics).

Ideally, some help for translating the formal specification back into natural language should be available, such as the GIST paraphraser mentioned above, in order to ease understanding by people other than the developer, in particular the customer.

A more formalized approach, stating proof obligations and perhaps even discharging them in formal correctness proofs, can be very useful when there are (very) high demands on correctness and reliability of the software system developed, or at least central components of it. This tends to be the case in systems software and, even more so, in safety-critical systems such as nuclear power stations, weapon systems, air or rail traffic control. Nevertheless, developing software as part of a system where it is not allowed to fail (because of the large potential damage) remains irresponsible, even when using comparatively "safe" approaches such as formal methods.

For formal methods to be useful, some form of quality system should be in place first, and the developers who are to use formal methods have to be specifically selected and trained for this task.

Even without actually using formal methods, the ability to use them can help develop better software, for example by asking oneself "How would I prove this program correct if I needed to" or by being able to write better assertions inside program (e.g. with the `assert` macro in C programs).

What does all this mean for the formal methods community? First of all, realistic presentations of its achievements are needed, based on an understanding of the limitations of formal methods and the willingness to admit these.

Concentrating on those areas where formal methods are most useful, formal methods teaching, consulting as well as marketing (outside the formal methods community), should be continued and, where possible, extended. So far, these efforts show little success. To overcome this problem, the current research effort on supporting slow evolution of development methods towards a more formal approach (combining formal, semi-formal and informal approaches), and making formal methods easier to use and less frightening for the average software engineer, should be extended. *Slow evolution* towards new methods such as formal methods is an absolute must in most industrial environments since otherwise the risk, due to an unstable development environment, and the cost of introducing them is too high. A lot of cultural change is needed before formal methods can be adopted, and this does not happen quickly. Researchers in the area of formal methods should therefore work on adding formality to the currently used informal or semi-formal approaches, without throwing away all the experience gained (cf. the work referenced in [LPT94]).

Another research area where more work is needed is the handling of complex language features and of the technical environment, e.g. by formally describing the semantics of a "real-world" programming language such as C or Assembler, which are often used in the kind of software where formal methods are most useful.

References

[BSH+93] F.C. Brodbeck, S. Sonnentag, T. Heinbokel, W. Stolte, and M. Frese. Tätigkeits-

schwerpunkte und Qualifikationsanforderungen in der Software-Entwicklung: Eine empirische Untersuchung. *Softwaretechnik-Trends*, pages 31–40, May 1993.

[CDH+96] Jean-Pierre Courtiat, Piotr Dembinski, Gerard J. Holzmann, Luigi Logrippo, Harry Rudin, and Pamela Zave. Formal methods after 15 years: Status and trends. A paper based on contributions of the panelists at the FORmal TEchnique '95 Conference, Montreal, October 1995. *Computer Networks and ISDN Systems*, 28:1845–1855, 1996.

[CH79] D. Coleman and J.W. Hughes. The clean termination of Pascal programs. *Acta Informatica*, 11:195–210, 1979.

[CNP+92] W. Coy, F. Nake, J.-M. Pflüger, A. Rolf, J. Seetzen, D. Siefkes, and R. Stransfeld, editors. *Sichtweisen der Informatik*. Vieweg, 1992.

[Fai85] Richard E. Fairley. *Software Engineering Concepts*. McGraw-Hill, 1985.

[Fet88] James H. Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[FLBG95] John S. Fitzgerald, Peter Gorm Larsen, Tom Brookes, and Michael Green. Developing a security-critical system using formal and conventional methods. In *[HB95a]*, chapter 14, pages 333–356. 1995.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[HB95a] Michael G. Hinchey and Jonathan P. Bowen. *Applications of Formal Methods*. Prentice Hall, 1995.

[HB95b] Michael G. Hinchey and Jonathan P. Bowen. Applications of formal methods FAQ. In *[HB95a]*, chapter 1, pages 1–15. 1995.

[HP95] Hans-Martin Hörcher and Jan Peleska. Using formal specifications to support software testing. *Software Quality Journal*, 4:309–327, 1995.

[Hum88] Watts S. Humphrey. Characterizing the software process: A maturity framework. *IEEE Software*, 5(2):73–79, March 1988.

[ISO91] *ISO 9000 Part 3. Quality management and quality assurance standards — Guidelines for the application of ISO 9001 to the development, supply and maintenance of software*, 1991.

[JJLM91] C.B. Jones, K.D. Jones, P.A. Lindsay, and R.C. Moore. *mural — A Formal Development Support System*. Springer-Verlag, 1991. With contributions from J. Bicarregui, M. Elvang-Gøransson, R. Fields, R. Kneuper, B. Ritchie, A.C. Wills.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall Int., 2nd edition, 1990.

[Kne91] Ralf Kneuper. Symbolic execution: a semantic approach. *Science of Computer Programming*, 16:207–249, 1991.

[LG97] Luqi and Joseph A. Goguen. Formal methods: Promises and problems. *IEEE Software*, pages 73–85, January 1997.

[LPT94] Peter Gorm Larsen, Nico Plat, and Hans Toetenel. A formal semantics of data flow diagrams. *Formal Aspects of Computing*, 6(6):586–606, 1994.

[Mar90] Lynn S. Marshall. Formally describing interactive systems. In Cliff B. Jones and Roger C.F. Shaw, editors, *Case Studies in Systematic Software Development*, pages 293–336. Prentice Hall Int., 1990.

[MLP79] R.A. De Millo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5), 1979.

[Nau82] Peter Naur. Formalization in program development. *BIT*, 22:437–453, 1982.

[PM87] H. Partsch and B. Möller. Konstruktion korrekter Programme durch Transformation. *Informatik-Spektrum*, 10:309–323, 1987.

[Swa82] William R. Swartout. GIST English Generator. In *Proc. of AAAI-82*, 1982.