International Journal on Software Tools for Technology Transfer Manuscript-Nr.
(will be inserted by hand later)

The Meaning of "Formal": from Weak to Strong Formal Methods

Pierre Wolper

Institut Montefiore, B28
University of Liège
B-4000 Liège Sart-Tilman, BELGIUM
Email: pw@montefiore.ulg.ac.be

November 3, 1997

Abstract. This short note reflects on what makes formal methods "formal". It concludes that there are weak and strong ways of being formal, the latter being linked to the formality of the method being exploitable, and exploited, in software tools.

1 Introduction

"Formal Methods" is becoming a widespread expression in software engineering. It refers to methods and tools that are supposed to bring to software development the rigor of mathematical reasoning and the certainty of correctness it implies. Formal methods are sometimes described as the "applied mathematics of software engineering". However, not everyone agrees on precisely what constitutes a formal method (as opposed to an informal one). Clearly, a formal method has to involve some mathematical notation, but is this sufficient? Most likely not, but what is then the criterion that defines a formal method? This is precisely the question that this note attempts to answer. It does so by giving a series of criteria that methods should satisfy in order to be formal. It then concludes that the essential requirement for a method to be formal in a strong sense is that it be supported by software tools that can perform a meaningful semantical analysis.

2 Formal Methods and Syntax

In the "formal methods" approach to system development, one views the development process as starting with a high-level description or *specification* of the intended behavior of the system. More and more detailed descriptions of the system are then produced from this specification until the actual runnable program is reached. Ideally, one would be able to check that the implementation that is finally produced satisfies all requirements expressed by the specification.

Correspondence to: Pierre Wolper

The first question that comes up when using this approach is the choice of a *notation* for expressing the specification. English, for instance, is not suitable. Indeed, it is too ambiguous to be used in a rigorous development process, to say it otherwise it is not "formal" enough. So, formal methods are essentially always centered around a *formal* specification language. What does this mean?

A very basic requirement for a language to be "formal" is that its sentences are defined in a precise way. In other words there is no ambiguity on what is or is not a sentence in the language. It is perfectly reasonable to strengthen and clarify this condition by requiring that a formal specification language have algorithmically recognizable sentences. In simple terms, the syntactic analysis of the language can be done algorithmically. The following will thus be our first criterion for defining formality.

Criterion 1 (Decidable syntax). *A language has a decidable syntax if its sentences are recognizable algorithmically.*

It is quite a weak requirement and, indeed, all existing formal methods use specification languages with a decidable syntax.

3 Formal Methods and Semantics

The next step in formality of a notation is to require that it has a *formal semantics*. In general, a semantics for a language is given as a mapping from that language to another, usually simpler, formalism. For instance, the semantics of a program can be given in terms of its set of possible execution sequences. When is such a mapping "formal". One tempting answer is to say that it must be algorithmically computable in the Turing sense. This is, however, too strong since it would lead to intuitively absurd conclusions such as claiming that the semantics of first-order arithmetic [End72] is not formal. Looking more closely at the problem, one sees that the semantics of formal specification languages are usually not decidable because of some quantification on infinite domains. So, the idea is to consider semantics to be formal if they are build from a algorithmic kernel to which quantification is added. But this is exactly the arithmetical hierarchy of undecidable classes for first-order quantification and the analytical hierarchy when quantifying over functions is allowed [HR67]. The following *formal semantics* criterion is thus proposed.

Criterion 2 (Formal semantics). *A language has a formal semantics if deciding semantical questions for this language (e.g. equivalence of sentences) is proven to fall within the arithmetical or the analytical hierarchy.*

Again, this is not a very stringent requirement and it is satisfied by virtually all languages used in methods that claim to be formal. Note that the requirement is that the semantical problems are *proven* to fall within the arithmetical or analytical hierarchies. This implies that not only must the fact be true, but also that the semantics should be defined a way that makes the proof possible.

4 Formal Methods and Semantical Analysis

The practical purpose of having a language with a formal semantics is to to make some semantical checks on specifications possible. For instance, it is important to have the capability of checking that a specification implies a given property or of checking the coherence of specifications at different levels of abstraction. However, according to Criterion 2, this could be a highly undecidable problem. There are several ways of working around this obstacle. The first is to restrict the type of system one is dealing with in order for semantics questions to become decidable. One common way of doing this is, for instance, to consider only finite-state systems.

However, decidability is not essential to obtain results. Indeed, a partial decision procedure, which is not guaranteed to terminate but which produces a correct result when it does, is also extremely useful. One does not even need a partial decision procedure in the strict sense (that always stops on positive instances) in order to have meaningful practical results. What really matters is that the semantical analysis can be carried out with the help of a software tool that requires little or no human intervention (if this is not the case, it is very unlikely to be performed). Note that a guarantee of always obtaining a result is not necessary and is moreover often illusory. Indeed, even when within a decidable class of system, an analysis often does not terminate due to excessive time and space requirements. To the user there is no difference between an analysis that never terminates and one that terminates in a billion years. Our last criterion is thus the following broadly expressed one.

Criterion 3 (Semantical Computational Support). *A formal method provides Semantical computational support if it allows software tools for checking semantical properties of specifications.*

This criterion is somewhat more fuzzy than the first two, but it is nevertheless clear that existing formal methods do vary widely with respect to it.

5 Classifying Formal Methods

Since virtually all formal methods do satisfy the first two criteria that have been given, the proposed classification is based on compliance with the third, which in the view of the author is essential for methods to be formal in a practically meaningful way. The following are thus distinguished.

- **Weak Formal Methods** (specification only formal methods), and
- **Strong Formal Methods** (formal methods with tool supported semantical analysis).

In weak formal methods, a language with a decidable syntax and a formal semantics is used to specify the system being developed. Tool support is limited to checking the syntax of the specification. The usefulness of these methods is that they force the system designers to think about what their system is supposed to do and to express it more abstractly than by a program. This is often beneficial, can lead to discovering errors, usefully documents the system design, and can serve as a means of communication between people involved with the various parts of a system. To take an analogy, it amounts to writing the equations describing a physical system but without attempting to analyze the solutions to these equations.

For a method to be in the "strong" category, it is required that at least some form of semantical analysis is possible with the help of mostly automatic tools. Again using the physical system analogy, it means that some software package for solving the equations is provided. It does not imply that the software package can handle any system, just that it often enough provides useful information to the system designer.

The terms "weak" and "strong" might appear to have a value connotation. This is intentional. Without semantical analysis formal methods are of very limited value with respect to their stated goal of ensuring the correctness of software systems : their formal syntax and semantics are just theoretical properties, not assets that are exploited in a substantial way. From the point of view of the author, a strong formal method even with limited applicability is more meaningful than a weak one that is perfectly general.

The reader might be wondering where formal methods in which proofs are done by hand fit into the weak/strong classification. The answer is in the weak category. Indeed, for any nontrivial system, the proof will just never be done and, if ever it is, it is unlikely to be read, which makes it quite unreliable. Finally, a few words about the fact that no examples or formal methods are given in this note. The reason for this is that the situation is evolving and methods that have long been weak are gradually evolving towards being strong methods. Given this trend towards strong methods, a classification done now might seem quite unfair in just a few years.

References

[End72] H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, 1972.

[HR67] Jr Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.