Formal Methods and their Evaluation

Joseph Sifakis
Projet Spectre - Verimag,
2 Rue Vignate,
38610 Gières, France
e-mail: sifakis@imag.fr

1 Introduction

Producing systems which satisfy given quality requirements at reasonable costs and within reasonable delays is an important economic stake and also a scientific and technological challenge.

To respond to this challenge, it is necessary to have methods and tools for :

- predicting/guaranteeing/evaluating the quality of the developed systems
- increasing the efficiency during lifecycle

The application of *formal* methods, that is, methods relying upon the use of mathematical synthesis and analysis techniques, is considered in principle, as the basis of a rigorous development approach. However, the effective exploitation of these methods encounters several problems that can be explained by their inherent limitations (undecidability, intrinsic complexity), the increasing complexity of the applications, the lack of users with appropriate skills and background and finally, the lack of theoretical results.

In spite of these problems, the strong requirements concerning the quality of critical applications makes mandatory the application of formal methods in the development process. Furthermore, the wide use of computerized systems strengthens and extends criticity requirements in an increasing number of applications areas such as nuclear plants control, transportation, avionics and space but also, networks, telecommunications and banking.

This paper tackles the problem of the applicability of formal methods and of methods in general, and proposes a framework for their evaluation. It starts with a presentation of the formal methods application in the development process. This presentation is not at all exhaustive and is used in order to introduce terminology and to describe the main existing approaches. There exist many surveys on formal methods and development methods in general, that the reader can consult to learn more about the languages, methods and tools mentioned.

The second section proposes a general framework for method evaluation using four classes of criteria. As an application of this evaluation methodology, we draw some general conclusions about the appropriateness of the existing formal methods and point out some important work directions. The evaluation methodology presented has been studied and applied under ESA/ESTEC contract “DMS design validation”.

2 The application of formal methods

2.1 The top-down development paradigm

Ideally, the development process can be considered as a chain from the initial requirements and the formulation of a problem to its solution and implementation reached by application of successive transformations. Along this process, it is necessary to have languages, methods and tools to guarantee the conformance of the obtained descriptions to the initial requirements. Concerning languages, description at different levels of abstraction and in different styles are necessary. One can distinguish at least three types of languages :

- *requirements description* languages, which are usually informal languages used by non specialists to express initial requirements
- *specification* languages, used to express system behavioral descriptions
- *programming* languages which are executable languages; requirements description and specification languages are not executable in general.

To maintain the coherence in the overall development process, methods and tools are used :

- either to derive a description from a given one, by synthesis, translation, transformation, compilation. It is desirable that the semantics of the description is preserved or narrowed by derivation.
- to check conformance of a description or implementation obtained at some stage of the development process with respect to some description of reference.

  The term *verification* is used when checking is performed by comparison of a description against requirements about the behavior (sets of values, of traces, of models). Verification methods are performed by analysis of a description. They cover a large spectrum of checking activities going from informal code inspection methods, to formal verification that a description satisfies a property or that one description is a refinement of another.

  The term *test* is used to denote checking of the *observed* system's behavior either real or simulated. To apply a testing methodology, we assume a distinction between inputs (controllable events) and outputs (observable events). Test cases define strategies of application of inputs in terms of the observed output, in order to check that a requirement is satisfied. Usually verification methods are applied upstream of the development chain while testing is applied downstream. Clearly, implementations can be validated only by testing.

2.2 The state-of-the-art

The actual application of formal methods is restricted to fragments of development activities. It concerns essentially activities such as specification and verification and to a lesser extent programming, testing, and the formulation of requirements.

2.2.1 Specification

One can distinguish two main approaches according to the type of language used :

- Logical specification languages which are based on the use of *logics* such as B, Z, TLA, VDM, RAISE, Unity and TLA. They are the most general languages as they allow the description of properties and behaviors. The logics used are either general predicate calculi or special program logics adapted to the expression of system properties such as temporal logics.
- Behavioral description languages that allow to specify system behavior as the composition of behaviors of elementary operators. Their semantics is often expressed in terms of transition systems modulo some abstraction criterion. Typical representatives of these languages are protocol specification languages like SDL, Lotos and Estelle, synchronous languages like Esterel, Luste and Signal as well as formalisms based on the use of automata such as statecharts and Petri nets.

The main features of the two approaches are the following :

- Logic based languages are well adapted to the expression of global properties such as termination, correctness, absence of deadlock, mutual exclusion, fairness. An important feature is their abstraction. Specifications are conjunctions of formulas and their meaning is the intersection of their characteristic sets. They can be easily modified by removing or adding formulas. However, specifications can be inconsistent, incomplete, trivial, redundant.
- Behavioral specification languages are less abstract but easier to use. Abstraction is achieved by hiding actions or variables. Their main advantage is that they are executable. Furthermore, if they support appropriate abstraction mechanisms they allow a direct expression of properties difficult to express with logics like causality relations.

2.2.2 Verification

One can distinguish two main approaches :

- An approach based on the use of an inference system which allows to establish conformance to specifications by reasoning on the structure of the description. Their main weakness is that they require interaction with a skilled user. This approach applied to behavioral or logical specifications is naturally compositional in the sense that properties of a composed system are inferred from properties of their components.

- An approach based on the use of models which consists in extracting from a description a model representing explicitly or implicitly a transition relation. This approach has been successfully applied to finite state systems or to systems which admit finite abstractions such as systems with complex control structures and whose behavior is "data independent". The advantage of this approach is that it is fully automatic and requires minimal user interaction. Its main limitation is the complexity of the models generated.

2.2.3 The two frameworks

Research on formal methods is developed in two complementary frameworks :

- The "single language" framework based on the use of general powerful languages such as predicate calculi. This provides a uniform framework for development and verification. Programs, properties or behaviors, have the status of formulas; semantic relations such as equivalences, refinement, implementation, satisfaction relations are expressed as relations between formulas. The verification method often used in the framework is by inference system and the tools are theorem provers or theorem checkers as for example, HOL, PVS, Coq, and B-tool.
- The "multilanguage" framework which combines the use of domain specific languages for example languages for the description of hardware, of protocols, of real-time systems, and of logics such as temporal logics. The use of specific languages enhances expressivity. On the other hand, the absence of a unified framework for the expression of programs and their properties can be considered as a limitation. In this framework, model-based verification tools are often used such as SPIN, Aldébaran, Cospan, SMV.

It is important to emphasize that the two frameworks are not incompatible. The "single language" framework allows a better coverage of upstream development phases while the other of downstream development phases. Current research focuses on the unification of the two frameworks : behavioral description languages are integrated in universal language frameworks and algorithmic decision methods are used in theorem provers.

3 The methods and their evaluation

3.1 The methods

A prerequisite to any discussion concerning the advantages and the interest of methods is a clear understanding of what methods and their main characteristics are. We introduce some notions to provide a basis for method comparison and evaluation in the context of technological processes such as the system development process.

A method is often defined as the conceptual approach to the solution of a problem. The problem can be specified as a relation P between initial data from a domain D, and a domain of results R. A method M_P solving the problem P, can be considered as a "recipe" describing *how* from any initial data d of D a result (solution) r of R can be derived such that dPr (if a result exists for d).

Ideally, a development method is a method which from given initial requirements derives a system which satisfies the requirements (the development problem is defined as the satisfaction relation relating requirements to implementations). As explained in the previous section, development methods make use of other methods which can be decomposed into simpler methods etc. The following table shows such methods with their input and output domains :

input	*problem/method*	*output*
specifications	synthesis	program
specifications + program	verification	{yes, no}
program	compilation	executable code
system + test purposes	testing	test suites

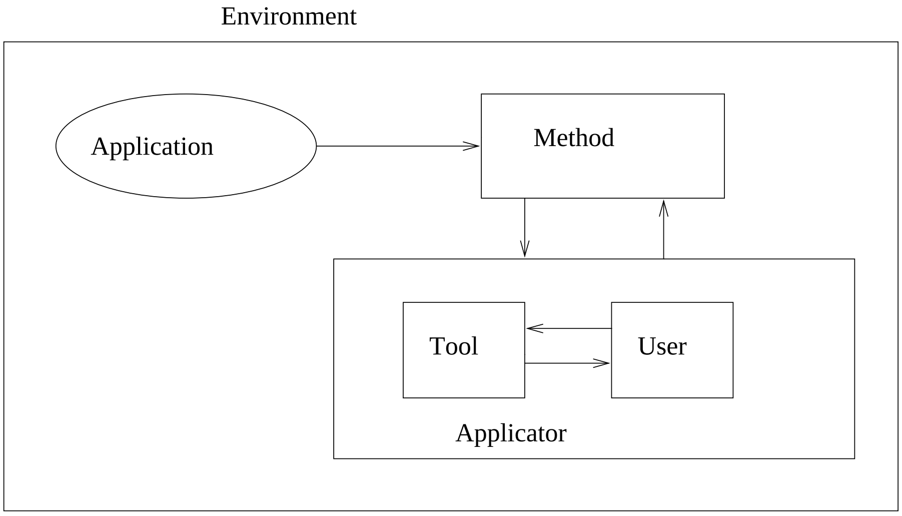
An important fact about the methods considered is that they are intrinsically hard either because they address problems without algorithmic solutions or because of their complexity. For instance, program synthesis from logical specifications and program verification are in general, undecidable problems while compilation has algorithmic solutions for most of the executable languages.

A discussion on the nature of the methods is certainly beyond the scope of the paper. However, we believe the following characterize a method :

1. The existence of a *language* (universe of discourse) whose terms are used to derive results from initial data.
2. The possibility of *checking soundness* that is, that a result produced by application of the method is a solution to the problem that the method is supposed to solve.
3. The possibility of applying the method in a more or less systematic manner.

According to the type of language used, a method is called *formal* if both the syntax and the semantics of the language are formally defined, semi-formal if the syntax is formally defined and *informal* otherwise. It is important to notice that very often, a method is reduced to the use of a language.

The language contains the primitives, constructions and concepts available to the user of the method. Methods are defined as a set of derivation rules



in the language specifying computation steps. Clearly, the choice of the language determines implicitly the granularity of the computation steps, the style of description, the nature of solutions. Algorithms are methods using computable primitives and terminating within a finite number of steps; they are fully mechanizable methods. Non algorithmic methods can be described as algorithms involving non effective computation steps. The latter can be decomposed, without loss of generality, into a *guess* part which is an undecidable existential predicate on data and a computable *action* part. The guess part corresponds to a test or a precondition for the application of the step while the action describes the effect of the computation.

An important consequence of the above distinction is that non algorithmic methods involve non mechanizable computation that can be ideally performed by some "oracle" and a mechanizable computation that can be supported by tools. The oracle task requires inventiveness and intelligence. It can be carried out by a human user (possibly assisted by machine). This distinction is an important one. It reflects the fact that a method is applied by a pair user/tool that we call *applicator* of the method, in the sequel. We believe that efficient application of methods requires some optimal division of work between the user and the tool, the user being responsible for critical decisions and the tool performing computational tasks. Extreme cases appear when one of the two components of the applicator is missing by giving *fully automated* methods and *manual* methods.

The second characteristic is an essential quality of any "scientific" method. It implies the possibility of validating method application with respect to correctness criteria. For informal and semi-formal methods, soundness can be checked experimentally (tested) : for a given sample of initial data one can check whether the applicator of the method obtains correct results. This increases the confidence in the method but does not exclude erroneous results. Only formal methods can be guaranteed sound by establishing correctness for any initial data.

The third characteristic expresses a requirement of efficiency in the application of a method. It is more difficult to define as it depends on subjective factors. The percentage of success in the application of a method for a sample of users of reference, or the time necessary for learning the method could be used as qualifying criteria. Another criterion is certainly the existence of tools. The latter play two complementary roles : they allow to improve efficiency and prevent application errors as they "encapsulate" sub-methods.

It is remarkable that many papers reporting the application of formal methods emphasize the impressive achievements of mathematically strong researchers rather than the elements of a method susceptible of being widely applied in a development process. Not enough emphasis is put on methodological aspects distinguishing what can be actually systematized and what is simply possible. Crossing the atlantic ocean in a canoe may be of anecdotal interest but cannot be a basis for the development of transatlantic maritime transportation.

3.2 Method evaluation criteria

Method evaluation should take into account both intrinsic method characteristics and the concrete exploitation conditions. We define hereafter 4 criteria for method evaluation.

3.2.1 Matching of the Method to the Application

Applications differ in their nature and complexity. It is important that the method proposed for a given application matches the characteristics of the application. Mismatch can result if the method and its associated language are too weak to capture conveniently the behavioral aspects of the considered system. For example, some methods were originally designed to handle sequential programs whose semantics can abstractly and naturally be described as a relation between the initial and terminal state of a computation. It is a mistake to apply such methods to reactive systems whose natural semantics are sets or trees of computations.

On the other hand, a mismatch can also result from applying heavy formalisms to a too simple application. For example, it would be inappropriate to use temporal logic for the specification and verification of sequential programs. The characteristics of the applications concerning the complexity, the type, the quality and other requirements constitute an important class of criteria for the adequacy of a method.

complexity: Depending on the application domain, the size of applications can vary considerably. For example, embedded real-time applications are usually more than one order of magnitude less complex than telecommunication applications. Methods applicable to systems of medium or small size are not necessarily applicable to large size systems.

type: The type of the application concerns features of the control, the data and the underlying execution structure. It is difficult to make a classification of these interdependent features.

- **synchrony/asynchrony:** Some applications can be strictly synchronous, for example hardware, while others asynchronous, for example information systems. However, in most cases it is desirable to combine the two modes of functioning.
- **reactivity:** Some applications are designed as purely transformational systems that realize an input/output relation while others as reactive systems that interact with an environment that they control. In the latter case, an important assumption is reactivity which means that the controlling system is fast enough to consistently take into account all the changes of the environment. To achieve reactivity, it is necessary to adopt some synchrony assumption and use *time* to express constraints relating relative speeds of processes.
- **distribution:** Distributed applications require methods that allow the description of asynchronous control and communication mechanisms. Strictly synchronous languages are not well-adapted for the description of distributed systems. The problem of combining reactivity and distribution is not completely solved at theoretical level.
- **data modeling and handling:** According to the type of application the characteristics of data vary. For digital circuits booleans are sufficient, for real-time control applications simple data types while for other applications powerful data handling and structuring methods are necessary.

quality and other requirements: The quality of an application can be characterized by performance requirements, dependability requirements such as, reliability, safety, security, maintainability and other qualitative requirements concerning the service provided.

Other requirements concern the life cycle of the application itself such as extensibility, reusability, genericity. These requirements are less crucial for custom-built applications while they get important for mass production.

3.2.2 The Applicator

The applicator is the pair user/tool that cooperate in the application of the method. Its characteristics depend on the capabilities of each component separately but also on the synergy of the cooperation. We distinguish the following criteria.

human factors: This criterion concerns the adequacy of the method with respect to the competences, the experience and the cultural background of the users. It can be characterized by the ease of learning, the training, the readability of the languages used.

The use of a rigorous language based on advanced concepts can be source of errors and frustration for non trained users. To achieve adequacy one can act on two complementary parameters: improving competences by training and adapting the method to the users level by developing appropriate languages and tools.
There exist currently divergences on the relative importance of these two parameters. Some experts claim that training is much more important than tools; given the nature of the methods, tools will play a rather marginal role of checking that a human correctly applies the method, the inventive work being carried out by highly qualified humans. Others stress on the necessity of adapting methods to the users level by specialization of the results.

Tooling: A tool allows to solve a class of problems that are involved in the application of the method. Several criteria determine the quality of a tool such as :

- The integration of the state-of-the-art which in principle, determines how completely the tool supports the method.
- The performance of the tool. Clearly, the same method can be implemented more or less efficiently depending on the design of the algorithmics and the processing capabilities.
- The scalability of the method and tool, that is, do they support large project developments in terms of application size, team size, subcontracting and geographical spread.
- The tool openness and extendability, that is, its ability to be integrated into existing environments, the existence of useful import/export formats.

Tool/user interaction: Two kinds of criteria characterize the quality of tool/user interaction :

- The user-friendliness of the tool including, quality of graphical and textual editor, user guidance, documentation capabilities, automation of tool operation, ease of parameter changes, ease of conceptual changes, V&V support.
- The user support in critical decision making which includes the generation of reports, diagnostics, debugging support, decompilation etc...

3.2.3 The Method

Relevant characteristics of a method are :

The underlying language features which include the following:

- **Level of formality:** It is generally argued that the level of formality is related to the ease of description : the more a language is formal the more it is difficult to describe. However, this argument does not take into account the "quality" of the description even if an informal and a formal description have the same "information content".
- **Level of abstraction:** Describing at different levels of abstraction implies the existence of sufficiently powerful description and structuring primitives. The latter include the possibility of composing and encapsulating descriptions as well as the possibility of refinement of both data and behaviors.
- **Evolutivity:** This feature concerns the possibility of modifying or extending descriptions in an incremental manner and is related to the possibility of modular, compositional, parametric or generic descriptions.
- **Reusability:** This feature concerns the possibility of reusing already defined components and is related to the possibility of modular, parametric or generic descriptions.

Checking possibilities: These criteria are not independent from the language features and concern the possibility of checking various kinds of properties.

- **Syntactic checking:** This type of checking is possible for semi-formal and formal methods.
- **Semantic checking:** This type of checking includes the verification at compile time of some general properties of the descriptions, such as consistency.
- **Functional/Behavioral verification:** This includes comparison of descriptions against logical properties or behavioral specifications.
- **Performance evaluation:** This includes both probabilistic and deterministic analysis of quantitative properties concerning timing, delays, throughputs.

Coverage of lifecycle: The spectrum of a method determines the possibility of carrying out a development without discontinuity and crossing smoothly the borders between the different types of activities.

- **Requirements:** They include the services provided by the system to all its potential users. They are often expressed in informal languages.
- **Specification:** Features of specification languages include specification of functions and operational modes, hierarchical representation and decomposition, abstraction, modularity, genericity of descriptions, traceability with respect to requirements.
- **Design:** Features of design languages include description of design components, hierarchical representation and decomposition, traceability with respect to requirements or specifications, facilities for testing and integration.

Codesign: This requires languages allowing the description and combination of both hardware and software components.

Code generation: Important selection criteria are efficiency of the generation process and optimality of the code generated. Other criteria are user assistance and guidance as well as design/implementation transition support.

3.2.4 Integration in the Environment

Methods evaluation should take into account the conditions of their integration in the environment which induces technological, industrial, commercial and institutional constraints.

Development environment: The characteristics of the environment where the method is integrated, play an important role for its exploitability. A development environment is composed of various software and hardware components : compilers, code generators, simulators, platforms, configuration and version managers, libraries. The integration of a method requires compatibility with the other methods and tools: functional compatibility for the interconnection with other components and, performance and scaling compatibility. The latter is important as the capabilities of a development process are limited by the capabilities of the less efficient sub-process.

The integration problem is especially important for formal methods that are usually applicable to intermediate stages of the development process. Even though the scope of application of formal methods is extending upstream, there will always exist a border between formal and informal techniques. Thus, the question arises of the connection between them (smoothness of the transition for instance).

Institutional/Industrial/Commercial Context: These criteria concern integration features external to the development environment.

Institutional The integration of a method in an international standard or its recommendation by an organization is certainly an important criterion as it implies an important community of users and a potential interest for the development of tools supporting the method.

Commercial The existence of commercially available tools that are maintained and improved following the state of the art is a guarantee of continuity and contributes in minimizing the risk of using a method. Clearly, the industrial exploitation of methods cannot be supported by prototype tools and is risky when there exists a unique supplier.

Industrial The industrial exploitation with recognized benefit is the ultimate proof that the method is interesting for the considered application domain.

3.3 Discussion

The application of the evaluation criteria to existing methods and tools shows that currently available methods fail satisfying most of the important criteria such as coverage of the lifecycle, scalability, integration in existing environments.

The current situation in industry is characterized by the fragmented use of a large number of semi-formal methods and tools. Formal methods application is marginal, in general. Development methodologies fully supported by tools exist for circuit design and medium size time critical applications. In most cases, tool support is fragmented due to the lack of compatible (not necessarily formal) tools and methods. As a rule, their is no direct relation between the implementation and models developed to validate specification or design choices.

The top-down development paradigm is too simple to be applied to complex systems development. For complex systems, the solutions adopted are based on already existing implementations by reusing components, architectures or by simply extending the functionalities of existing systems by ad hoc techniques. Such an approach is the only realistic for both economical reasons (using the cumulated competences of teams) and in order to master the complexity by incremental modifications. Tool support in this case, requires a perfect integration of methods and tools used at different stages of the development process with in particular, the possibility of combining descriptions and tracing back the impact of decisions at different description levels.

The needs for efficient methods and tools for the development of complex systems are spectacularly increasing. They concern all the lifecycle activities .

Currently, there are no rigorous languages for the expression of requirements. The underlying semantics of object-based formalisms supporting complex data description, reusability and powerful control mechanisms are not still well understood.

Distributed systems is certainly an area where there is a need for rigorous methods. The emergence of standards like ODP, CORBA, OMT should trigger research on the formalization of the underlying concepts, the semantics of the languages and the distributed implementation techniques.

System developers are concerned with eliminating errors in specifications and designs of systems; many errors result from incomplete analysis of the interaction of a system with its environment. Modeling and validation become important activities and the development teams lack skilled engineers to ensure them. Also, there is a clear need for conformance and interoperability testing methodologies taking into account problems of observability and controllability.

In spite of the current deficiencies, the need for rigorous development makes the application of techniques based on formal methods necessary. Their industrial application depends not only on the relevance and soundness of the methods but also on technological, sociological and economical factors. The proposed framework should help evaluating globally the appropriateness of methods.