Formal Description Techniques and Software Engineering: Some Reflections after 2 Decades of Research

Juan Quemada

Dept. of Telematic Engineering, Universidad Politécnica de Madrid
Ciudad Universitaria s/n, 28040, Madrid, Spain
jquemada@dit.upm.es
http://www.dit.upm.es/~quemada

Abstract. Software engineering is based today to a large extend on rapid prototyping languages or design environments which are high level, very expresive, executable and enabling the quick production of running prototypes, whereas formal methods emphasices the preciseness and proper mathematical foundations which eanble the production of unambiguous references needed in protocol engineering. The goals of formal methods and rapid prototyping are not in contradiction, but have very rarely been considered together. This paper analyzes the evolution, background and main divergence points, in order to highligh how convergence could be achieved.

1 Introduction

Mathematical models and techniques are at the core of many engineering disciplines and physical sciences. Those mathematical models usually define abstract views or properties of systems allowing a better understanding of the main parameters and elements. Traditionally, engineering disciplines have made use of mathematical models to highlight the relevant parameters of a given design problem, while hiding the irrelevant aspects to reduce the complexity of the design process.

Computer science and engineering differs from most engineering disciplines because it focusses in the design of digital systems which are discrete, as opposed to the analog nature of the systems addressed by most other engineering disciplines. Telecomunnication engineering dealt originally with analog radio or electrical signals, but the strong trend in last decades towards unification of information representation in digital multimedia formats has transformed most telecomunication systems into highly specialised computers for switching or processing some kind of multimedia information in digital format. For example telephone exchanges or packet routers. Therefore telecom engineering deals today also to a large extend with digital systems.

Digital systems are implemented directly as hardware systems when the complexity is low and the higher cost of the design is justified by large productions. But most digital system designs are implementated as software. Software based systems have usually a huge complexity and therefore research has focussed intensively during the last decades in methods and techniques able to cope with complexity.

Mathematical models of discrete systems have been used since the beginnings of computer science. Digital systems are modelled with various types of finite state machines, also called automata. But the huge number of states that most systems have, makes this model more a conceptual tool than a real engineering tool. Therefore automata were extended with standard programming variables to achieve a more understandable representation of large state spaces, leading to “extended automata” as a more powerfull mathematical model of discrete systems.

The programs of the first processors were coded directly in the machine language of the processor. The reference which defined the semantics of the machine language and the programs was the processor itself and there was not a need of a mathematical model of the processor for program designers.

High level languages, such as Fortran, Cobol, Pascal or C, appeared soon and provided mathematical abstractions of digital states in the form of variables and of program control in the form of high level program instructions. High level languages have a much higher expressive power than machine language and allow more productive and effective designs of programs. High level language programming is based on a set of software tools (compilers, debuggers, etc.) which allow execution of high level programs. High level languages have been also applied to the design of hardware systems.

High level languages created the need for new mathematical models, because tools for high level programming languages have to be implemented usually in many different processors and operating systems. Therefore a precise definition of the syntax and semantics of programming languages was needed, as a reference for compiler implementation, because all compilers should generate a code executing in the same way in each different processor or operating system.

There exists also a large community of more practmatic computer scientists that claim that the most effective way of defining semantics is having reference implementations. Reference implementations are implementations which have been extensively validated and agreed within a given community or committee to be the reference towards which correctness will be determined. Reference implementation are usually based on open software to facilitate product generation.

About two decades ago a big community of researchers and engineers started to apply mathematical modelling languages as a means to precisely define the semantics and behaviour of complete computer systems or parts of them, because they claimed that many problems of existing software or systems were due to the lack of a precise mathematical definition of the languages, procedures and tools used.

This community was especially important in computer networking [8], because network protocols are algorithms which have to be implemented in any machine to be connected to a network and mathematical models were considered the most precise way of specifying the protocols which should form the reference architecture of the standardized computer networks which were being designed at that time.

This community was named the “formal methods” or “formal techniques” community and had as it main objective the development of rigurous and precise mathematical techniques able to support the development of programs, computers systems, communication protocols, etc. The ultimate goal of this community was the development of a complete mathematical formalization of the software and systems design process, covering from the initial requirements specification phases to the final implementation of the running systems, which assured the correctness by construction of the implementation with respect to all the requirements and design decisions imposed during the development process.

To achieve this goal many new elements are needed such as, precise description languages, abstraction, stepwise refinement, correctness verification and validation techniques, transformation techniques, implementation generation techniques, testing tools and theories, etc. All this new developments should have a well defined mathematical semantics and should enable a new era of rigorous and fail save software and systems design.

The paper will focuss in the rest on the analysis of the achievements and failures and especially in the relation with techniques which have been accepted in industry for performing software engineering. The paper focusses also only in the use of formal methods in protocol engineering, communication networks and distributed applications, although many of the conclusions can be applied to a more general context.

2 Software Engineering and Formal Methods

Software engineering is the discipline concerned with creating and maintaining software applications by applying computer science, project management, domain knowledge, and other skills and technologies. Cost-effectivenes, product development lead-times, existance of proper design tools or environments and availability of trained people are mandatory issues for industry to accept new methods or tools.

Most protocol implementations are software developments and therefore, formal methods research should address industry priorities and should fully integrate into software engineering practices to be successfully incorporated. Lets analyze how formal methods and software engineering have evolved to try to understand better how formal methods research should be incorporated in software engineering practices.

Formal methods based processes rely on the vision that the main characteristics and features of a system can be specified in the first phase of the design process and that the rest of this design process refines this initial specification introducing design decisions which lead at the end of the process to a correct implementation which fulfills the initial requirements.

The design process may consist of more than one step, where each step takes as input a given partial definition of the system and generates a more complete definition of the system which should be proven correct with respect to the previous design steps, by some kind of mathematical correctness proof. In networking architectures the input specification is usually called the service and the implementation of the service is called the protocol.

This design model is very much in line with the waterfall model proposed in 1970 by W. Royce [1], which is considered somehow obsolete. It has been considered by several authors as the “dream of the western manager” because it would allow managers to precisely specify their objectives, strategies and requirements, which the rest of the organisation should just implement. Such a model would provide the project manager with an absolute amd rigorous control of the developments made.

During all those years of intensive research in formal methods, the software engineering community and industry has evolved and developed different approaches which have proven very effective, such as rapid prototyping, the spiral model, extreme programming, agile methods, etc [2, 3, 4], which have been widely accepted by industry.

Those methods are based on the vision that the main features of a complex system can not be properly understood in the first phases of the design process. Designers know at the beginning only the problem they have to solve.

The design process should be therefore an incremental learning and design process based on rapid prototyping. Early prototypes must be produced of the least understood parts, to gain a better understanding, as well as to obtain early user/customer feedback and allow testing.

As the “Manifesto for Agile Software Development” [3] states, the emphasis is put, in those approaches, much more on frequent software prototypes, adaptation to changes and direct interaction/collaboration among designers and/or customers, than on requiremments, planning or documentation.

In prototyping based software engineering approaches the emphasis is put on rapid prototyping languages or design environments which are high level, very expresive and executable, enabling the quick production of running prototypes, rather than in preciseness or proper mathematical foundations as formal methods emphasice.

The goals of formal methods and of rapid prototyping are not in contradiction, but have very rarely been considered together. The formal methods community should probably take into account the main trends of software engineering and try to provide solutions for the problems that software engineering has, rather than trying to develop a complete independent design framework.

2.1 Dealing with Complexity and Reusability of Software

Management of complexity has been one of the main challenges in software engineering. Abstraction is the main conceptual tool for dealing with complexity and most programming and specification languages include abstraction mechanisms such as, procedures as abstractions of operations, variables/records/structures as abstractions of state, objects as abstractions of program modules with clear usage interfaces, processes as abstractions of behaviour, etc.

There exist a large consensus that the object oriented model is the right abstraction mechanism in sequential programming languages, for building well structured programs, as well as reusable libraries of software components. On the process side there is not such a consensus and several models exist especially for interprocess communication.

Todays design and programming languages, as well as, software engineering tools have evolved to support the needs of both, software engineering practices and mangement of complexity. Nevertheless, the abstraction mechanisms supported in programing languages provide only syntactic support for the abstraction mechanisms.

Formal methods should have provided semantic support for abstractions and have produced interesting results in this direction, but the languages and mechanisms used do not fulfill the software engineering needs.

Providing support for semantic abstraction in a framework which is applicable in todays software engineering practices is one of the still unrealized main promises of formal methods.

3 Protocol Engineering and FDT Standards (Formal Description Techniques)

The advent of computer networking led to the proposal of protocol engineering as somehow different form software engineering [5, 6, 7, 8]. Protocol engineering considered the following vision and goals, especially within the community which considered that the use of formal methods was the only way of creating a rigorous engineering discipline.

- Protocol standards should be legal or defacto standards which provide an unambiguous reference for deriving implementations, as well as conformance test which could assess in practice the correctness of implementations with respect to the protocol standard.
- Protocols standards should be correct. As correctness can only be determined with respect to a given set of requirements, the service definition was considred the requirements to be met by a given protocol.

This vision and goals led to some specific challenges which have guided researchers in the formal protocol engineering community during the last two decades, such as

- Challenge 1. Development of a language for unambiguosly representing protocol standards: To achieve this challenge FTDs (Formal Description Techniques) should be developed and standardized to provide a unambigous means for describing protocol standards.
- Challenge 2. Protocol representations should be proven correct: To achieve this challenge each protocol should be accompanied by a service specification and a proof that states that the protocol is a correct implementation of the service. As there are some properties about correctness, such as deadlock or starvation absence, which are independent of the service specification an additional validation of such properties was required.
- Challenge 3. Protocols should also provide the best performance: To achieve this challenge automatic derivation of analytic or simulation models should be possible where protocol performance could be anlyzed and optimized.
- Challenge 4. Protocol representations should allow automatic derivation of correct protocol imlementations: To achieve this challenge automatic derivation of implementations using correctness preserving transformations were needed.
- Challenge 5. There should exist a procedure to verify or validate the correctness of protocol implementations: This procedure was assumed to be

based in conformance testing of implementations under test. To achieve this challenge automatic test suite derivation from the protocol description should be possible with sufficient coverage of the protocol behaviour and state space.

The first formal description technique was IBMs FAPL [5] which was used to deploy early IBM network architectures to a wide range of system, soon followed by other proposals. The advantages of having standardized FDT became clear soon. The first standardized FDT (or semi, becuase it was not fully formal at the beginning) was CCITTs SDL [6] which is based on an extended finite state machine model. SDL has been also the most successfull standardized FDT due to it's use for defining several CCITT/ITU standards, although the core of the software industry has not adapted it. The definition of the ISO-OSI reference model during the eighties and nineties led to the definition of two additional FDTs, which where competing with each other and with SDL as well. The first one was Estelle [6], which was based on an extended finite state machine model and standard Pascal data types. The second one was LOTOS [6], which was based on an algebraic calculus of processes and algebraic data types.

There were therefore 2 FDTs (SDL, ESTELLE) based on the less abstract “extended automata” model and one FDT (LOTOS) based on the more abstract mathematical theories of algebraic calculi of processes and algebraic data types. SDL and Estelle are much like programming languages and have more or less the same level of abstraction than C, Pascal, ADA or Java, although they were better suited for protocol representation. LOTOS on the other hand is more abstract, but it's main drawback for application in software engineering is the ACT ONE data definition language which has an algebraic semantics and is not executable. The behaviour part, based on a mixture of CCS [9] and CSP [10], was extremely powerfull and provided solutions for dealing with semantic abstraction which do not exist in todays design languages and tools. But the lack of executability of the data part made LOTOS difficult to apply.

Although protocols and network architectures have some minor distinguishing features with respect to other software developments, the mayority of the elements of the discipline are comon to software engineering and in my opinion, it would have been wiser to consider protocol engineering as a specialization of software engineering which inherits all it's elements and procedures. The design of the Internet was done following many of the software engineering principles explained before and its success was probably due to the higher effectiveness of rapid prototyping approach as compared to the more waterfall oriented approaches based on formal methods, which were used by standards organisations (ISO, CCITT/ITU) and the formal methods community.

4 Protocol Engineering and the Internet

The success of the Internet was due to many factors. The most important factor was probably the early availability of running implementations of the TCP/IP stack, as well as the availability of a large variety of applications. When ISO was starting work on developing FDTs, the Internet was already operational. Nevertheless, the development and of course the success of the Internet would not have been possible if the designers would not have provided effective solutions to the challenges of protocol engineering. The protocol engineering behind the Internet was not based in formal methods, but provided quite effective solutions which could align in many cases even with the agile software development manifesto.

The working procedures of the IETF, the Internet Engineering Task Force, where all Interent standards are produced since it was created in 1886, are close to the sofware engineering practices based on rapid prototyping described before. The working procedures of the IETF are also much more democratic than those of most standard organisations and have had a big impact in the way technology is produced today. The effectiveness of the procedures used for developing protocols and applications in the IETF led to the early availability of many running services, which had been properly tunned and adapted to users needs, even if many of the components used where not properly optimized. Lead times were more important than quality of the result.

The IETF promoted from the beginning the open participation of researchers into standardization committees, where participants could attend on a personal basis without any accreditation or fee as it is usually necessary in official standard bodies. IETF has also not avoided the existance of competing standards proposals, accepting only the proposals which were widely accepted by the user community. A standard was never accepted without two or more running implementations. Those implementations were used as references and were usually open software which could be used in the implementation of the standards on other machines. Those practices motivated a large community of researchers and developers to contribute to the production of the IETF standards.

The Internet designers dealt as follows with the challenges of protocol engineering

- Challenge 1. Development of a language for unambiguosly representing protocol standards: IETF standards are described as informal textual descriptions to facilitate the understandability. The use of ASCII text has been promoted to facilitate editing. Nevertheless, textual description of standards are complemented by reference implementations in C, Java, PERL, ..., which are the real references with which implementations must interwork.
- Challenge 2. Protocol representations should be proven correct: Correctness was substituted by rapid prototyping and user evaluation. Proposed standards had to have several running implementations interworking among them. User acceptance substituted proofs of correctness. This makes a lot of sense, because a correctness proof of a protocol implementation with respect to a service does not mean anything. The important issue is to have services which are accepted and usefull for users.
- Challenge 3. Protocols should also provide the best performance: Protocols were optimized using standard simulation techniques. Running prototypes provided also a lot of early feedback to improve the performance problems of the first versions of the standards.
- Challenge 4. Protocol representations should allow automatic derivation of correct protocol imlementations: Reference implementations did provide an effective way of deriving implementations, because most of them are based in open software. They were ported and easily recompiled in new machines.

Reference implementations were written in high level languages for which compilers existed in most machines such as C, Java, PERL, etc. Only the small part of the code which was hardware or O.S. dependant had to be rewritten.

- Challenge 5. There should exist a procedure to verify or validate the correctness of protocol implementations: Interworking of implementations substituted conformance testing. As most implementation were derived from the same reference implementation interworking was not difficult to achieve.

The solutions given to the challenges of protocol engineering were not very innovative, but were cost effective and ready to apply. Therefore innovation focussed in providing new services, new networking technologies or improved versions of the protocols. Most services were not optimized, but were providing a nice service, were running and were ready to deploy.

On the other hand, in the development of the ISO-OSI reference model and in CCITT/ITU FDTs were defined from scratch and as no agreement could be reached there were three FDTs competing. No tools were available at the beginning and a lot of time and research effort was necesary to have the first prototypes of the compilers and design environments ready. The first implementations of the protocols had to be therefore handcoded. In addition the ISO-OSI protocol stack had a lower performance than the Internet stack due to the fact that the protocols were first specified and then implemented. The design life-cycle was very in line with the waterfall model and did not assure a proper and well tunned result in time. When this was detected it was too late to produce a better version of the OSI protocols. There were many other causes for this delay, especially of political nature, but the use of a different methodological approach would have led very likely to a better technical result. The Internet had started deployment and as it was the only widely available working solution, it was adopted by industry despite of the big political support in favour of OSI.

5 Opportunities and Challenges for Formal Methods Today

Research in formal methods has not taken into account software engineering practices and methodologies as used in industry and therefore the results obtained are difficult to apply in real software developments. Software engineering practices need several features as mandatory, such as

- *Support for rapid prototyping.* The development of early prototypes plays a crucial role in todays systems design because it enables an early user or customer evaluation, which verification, validation or formal proof systems can not support by any means. Early prototyping allows to validate the usability, functionality or friendliness and enables early tuning or redisgn. Effective rapid prototyping languages must be executable and be very expressive:
    - o *Executability.* Non executable mathematical modelling languages do not seem to have applicability in todays software engineering because they do not allow prototyping.
    - o *Being high level.* Design languages must allow prototype development with a minimum effort and should have therefore powerfull instructions which

allow prototype implementation with a minimum number of instructions or statements.

- *Support reusability of classes and objects.* Design languages must facilitate reusability. As object oriented languages are considered as the ones which provide the best support for reusability, formal design languages should support object orientation.
- *Support for reusability of behaviour definitions.* The behaviour or process part of the existing design languages needs probably still substantial research, because no consensus exist about the best formalism. Algebraic calculi of process theory has provided executable process models with very high expressive power and nice abstraction features, although integration into conventional design languages should be done.
- *Support for semantic abstractions.* Design languages for complex systems need to have some kind of semantic abstractions which allow to decompose the design process into a sequence of understandable steps. Most design languages provide some support, but without providing a formal semantics, where abstraction mechanisms just perform syntax matching of interfaces. This is probably one of the places where formal methods can provide better design methods. For example the hiding operation of CCS [9] and LOTOS [6] is a very powerfull abstraction mechanism. The testing or conformance relations [11, 12] of CCS and LOTOS are formal notions of implementation which can be integrated quite smothly into software engineering practices.

Protocol implementations are like other hardware or software implementations and therefore protocol engineering should fully align with software engineering practices. The need for a more formal approach to systems and application design still exists, but must not ignore all the pragmatic lessons learned in large software developments in industry.

Formal methods researchers should try to develop design languages with support for rapid prototyping, with a high expresive power and also with support for semantic abstractions for classes and behaviours. This would allow a much smoother design process by stepwise refinement where, instead of having the usual sequence of non executable formal descriptions, a sequence of executable prototypes would be generated, where each prototype can be proven as a correct implementation of the previous one, but which can also be evaluated and validated by users/customers.

The LOTOS formal description technique [6] got very near to this approach at the behaviour part, providing semantic abstraction mechanisms which do not exist in the design languages used today in software engineering. But the algebraic data part made it unusable for software engineering. A language based on the LOTOS behaviour part and a conventional executable data typing would have been a very powerfull design language at that time. Some industrial trials performed in the nineties validated this approach [13]. It was a pitty that this opportunity was missed.

An expressive and executable language providing formal support to design by stepwise refinement can enhance todays state of the art. This language should incorporate all the features which today are mandatory in software engineering such as, object orientation or module and interface constructs, and of course should support semantic abstractions for objects and behaviour in a way which can be easily mapped in todays engineering practices.

There exist opportunities for using formal methods research results to enrich existing design languages and methodologies. For example: The new Web architectural framework with all the new XML based languages and tools; or to enhance well accepted design languages such as Java of C#.

References

[1] Royce, W.W., Managing the Development of Large-Scale Software: Concepts and Techniques Proceedings, Wescon, August 1970.

[2] Zave, P., The Operational versus the Conventional Approach to Software Development, Com. Of the ACM, 27 (2), 1984, pp.104-118.

[3] Boehm, B.W., Anchoring the Software Process, IEEE Software, July 1996, pp.73-82.

[4] Agile Processes, http://www.c2.com/cgi/wiki?AgileProcesses.

[5] Schultz, G.D., Rose, D.B., West, C.H., Gray, J.P. Executable Description and Validation of SNA, IEEE Transactions on Communications, April 1980, pp. 661-677.

[6] Turner, K. J. (Editor), Using Formal Description Techniques, An Introduction to Estelle, LOTOS and SDL, John Wiley and Sons, 1993, ISBN 0-471-93455-0.

[7] Bowman, H., Derrick, J., Formal Methods for Distributed Processing: A Survey of Object Oriented Approaches, Cambridge University Press, 2001, ISBN 0-521-77184-6.

[8] Special issue on Protocol Engineering, IEEE Transactions on Computers, Volume 40 , Issue 4 , April 1991.

[9] Milner, R., A Calculus of Communicating Systems, Lecture Notes in Computer Science 92, Springer-Verlag 1980.

[10] Hoare, C. A. R., Communicating Sequential Processes, Prentice Hall International, Englewood Cliffs, New Jersey, 1985.

[11] De Nicola, R., Hennesy, M., Testing Equivalences for Processes, Theoretical Computer Science, 34:83-133, 1984.

[12] Brinksma, E., Scollo, P., Formal Notions of Implementation and Conformance in LOTOS.

[13] Fernandez, A., Miguel, C., Vidaller, L., Quemada, J., Development of a Satellite Communication Network Based on LOTOS, IFIP Transactions C-8: Protocol Specification, Testing and Verification XII, pp. 179-193, North-Holland, June 1992, ISSN 0926-549X.