Int J STTT (2001) 3: 107–111 / Digital Object Identifier (DOI) 10.1007/s100090100050

The opinion corner

Verification is experimentation!

Ed Brinksma

Chair of Formal Methods and Tools, Faculty of Computer Science, University of Twente, PO Box 217, 7500AE Enschede, Netherlands; E-mail: brinksma@cs.utwente.nl, Internet: `http://wwwhome.cs.utwente.nl/~brinksma`

Published online: 15 May 2001 –

Abstract. The formal verification of concurrent systems is usually seen as an example par excellence of the application of mathematical methods to computer science. Although the practical application of such verification methods will always be limited by the underlying forms of combinatorial explosion, recent years have shown remarkable progress in computer-aided formal verification. This makes formal verification a practical proposition for a growing class of real-life applications, and has put formal methods on the agenda of industry, in particular in the areas where correctness is critical in one sense or another. Paradoxically, the results of this progress provide evidence that successful applications of formal verification have significant elements that do not fit the paradigm of pure mathematical reasoning. In this essay we argue that verification is part of an experimental paradigm in at least two senses. We submit that this observation has consequences for the ways in which we should research and apply formal methods.

1 A little history

The roots of formal verification lie in the observation, which broke ground in the 1960s and 1970s, that software programs can be seen as *formal*, i.e., *mathematical* objects. It led to extensive studies of formal semantical models for programming languages, and the development of (academic) programming languages for which nice formal models could be guaranteed to exist. This activity generated a deeper understanding of the structure of computer programs and provided the foundations for many of the mathematical tools and models that are known as formal methods today.

This text is an adapted version of Brinksma [2]

In addition to the development of mathematical models that would help to give a scientific foundation to programming, this period was also marked by a strong methodological interpretation of these achievements that promoted the view of programming as an essentially mathematical activity. If programs are mathematical objects and specifications of their intended functionality are properly formalised, then their correctness can be demonstrated by mathematical means. Much effort was directed towards the development of programming calculi in which the development of programs can be seen as a form of *equation solving*.

The beauty and strength of this vision were so compelling that they dominated the scientific research agenda for programming for many years. The movement as such acquired, perhaps inescapably, some ideological traits too, in the sense that it was less forgiving of practical programming as practised in industry, and the program validation methods used there, most notably testing. The cure for the diagnosed *software crisis* was thought to be found in the education of a new generation of academically trained programmers that would introduce the mathematical methods of programming into industry.

Although the mathematical school of programming has thoroughly influenced the study of programming, programming languages, specification, etc., it is now generally acknowledged that the mathematical theory of programming cannot be applied as was originally envisaged. Many arguments have been put forward to explain why it did not or could not work (see e.g., [10]), including circumstantial technical, economical, sociological, and educational reasons that we will not consider here. An intrinsic reason for failure that was initially overlooked is the, retrospectively, almost obvious fact that in general a correctness proof has a complexity proportional to that of the program involved. This causes direct problems in scaling up the method to deal with complex software sys-

tems, which was, of course, the ultimate goal. Proofs or derivations of such systems would be very complicated and therefore susceptible to errors themselves, directly undermining the essential contribution of the method.

The law of conservation of misery has made sure that there are no easy solutions to the problem. The term *formal methods* became commonplace somewhere in the 1980s to indicate the assorted formal notations, theories, and models that had been developed to help specify, implement, and analyse software systems. As by then not only sequential, but also concurrent and reactive systems could be mathematically described and analysed in terms of elegant mathematical models, a second wave of methodological optimism swept through academia and parts of industry. Having at our disposal methods for the formal description and manipulation of algorithms, concurrent interaction, and data, it was believed that the design and the development of a proof of correctness of complex systems could be a *shared* activity, known as *correctness by design*. Moreover, these integral *broad-spectrum* formalisms would be supported by powerful software environments to support the design and verification with the required precision, thus solving the problem of controlling the precision of complicated (or perhaps better: lengthy) formal manipulations. The failure of this second formalist attack on the software crisis was again due to many, diverse causes. Again there was one important intrinsic reason: the formal objects that corresponded to descriptions of (parts of) the systems designs were so large that they could no longer be manipulated effectively, not even by software tools. This phenomenon became known as the *combinatorial explosion*, or in the particular case of the explicit manipulation of system states as the *state-space explosion*.

2 Computer-aided verification

During the past few years, formal methods, and in particular formal verification, are again drawing the attention of the research community and (parts of) industry. This time it is not the result of a methodological movement, but the result of technological advances and research in the field of computer-aided verification. For the first time it has proved possible to formally verify parts of nontrivial systems with practical consequences. This success in scaling up verification methods from toy examples to small-size real-life systems is the first hard evidence that the use of formal methods can, under circumstances, be made consistent with the requirements of industrial engineering.

Broadly speaking, computer-aided verification can be categorised in two main streams: the *theorem-proving* approach, which uses tools to produce completely formal proofs of correctness, and the *model-checking* approach, which is basically a brute force approach to enumerate and check all reachable states of the system under verification. Both approaches can only work on the basis of a *model* of the system under verification. In theorem-proving the model is a *logical theory* characterising the (relevant) properties of the system. In model-checking the model must be operational so that systems states can be systematically produced, and usually takes the form an abstract program describing some sort of transition system.

Most of the arguments that we put forward in this essay apply to model-checking and theorem-proving approaches alike. Still, the reader may detect a certain bias toward model-checking techniques in their presentation, so that the general case for computer-aided verification is only obtained mutatis mutandis. This is due to the personal experience of the author, and also with the, at least currently, greater application potential of model-checking.

Some reasons for the growing success of computer-aided verification are:

1. The technological improvement of the necessary computing equipment. Because of the ever-increasing performance of systems in terms of speed and available memory (*Moore's law*), computations that were far beyond the possible ten years ago are a matter of routine today.
2. The development of means to curb the effects of the combinatorial explosion. *Abstraction* techniques are used to strip away information in the model that is not relevant for the verification at hand, and lead to a simplification of verification models. *Modular* and *compositional algorithms* apply a divide-and-conquer strategy to handle complexity by making formal manipulation local to well-defined parts of the model that are significantly smaller. In model-checking substantial progress has also been achieved by the use of clever *data types* that allow for compactification, such as BDDs and the use of hashing techniques.
3. The availability of serious *tool environments*. Much of the work on such environments has needed a considerable time to come to fruition, and effective tools have started becoming available now. The development of good tools requires sustained efforts over many years to develop stable architectures and profit from accumulating improvements.

Computer-aided verification tends to have a rather pragmatic approach. What can be achieved depends more on the capacities and limitations of the tools that are being used, and less on methodological considerations. Because verification problems in their entirety are too large to handle, the process is eclectic and concentrates on the essentials. This means that only crucial parts and properties of the system and its requirements, respectively, are formalised and verified. In practice, this means that computer-aided verification process is subject to a process of trial and error to determine how much of a system can be verified with the available resources.

As we are still in the early days of computer-aided verification, the knowledge about the effective scope of applications of the different tools and techniques is still very limited. Moreover, it appears to be difficult to generalise successful applications as we are sometimes confronted by *chaotic* behaviour, in the sense that small changes to a given problem may have big effects on the effectiveness of a particular verification technique.

Because of the substantial investments that must be made for computer-aided verification for industrial applications, typical examples concern systems whose correct functioning is critical in a certain sense. This is not restricted to the so-called safety-critical systems, but applies more generally to systems for which the abstract or real cost of their malfunctioning is too high. Highly replicated systems (partly) implemented in hardware are a case in point. Embedded systems in consumer electronics and the automotive industry are good examples, as well as the verification of hardware chip designs.

It is fair to say that computer-aided verification takes place in a context of experimentation. Different techniques are experimented with to increase the performance of the tools. Models and specifications are experimented with to see how much of a given problem can be verified. Typical examples of verification are found outside the world of pure software in interaction with more traditional engineering disciplines for which experimentation and measurement is an established method of quality control. In addition to this general experimental atmosphere that surrounds practical verification, there is another and more essential link between experimentation and verification.

3 Verification needs experimentation

Verification needs as its basis a formal model of the system that must be verified, the so-called *verification model*. As it plays a crucial role in the verification process, it can be said that a verification is as good as its underlying model. Obtaining valid verification models, i.e., models that faithfully represent the relevant properties of the objects they represent, therefore is a cornerstone of the verification process.

One of the strengths of the original paradigm of programs as mathematical objects is that a program text is (through its formal semantics) its own formal definition. The question of the validity of the formal model with respect to the reality of the physically executed program can be dealt with as a correctness requirement for the compiler (or interpreter) of the programming language. This has the advantage that the problem can be addressed and solved in generic terms, and the cost of producing a correct compiler can be amortised over all the programs that will be compiled.

As already indicated earlier, the situation for actual computer-aided verifications can be radically different. If a complete, formal definition of the system under verification is available then it will usually be too big to serve as an effective verification model. This means that additional efforts are required to obtain smaller models that are valid for the verification task at hand.

A way to approach this question is to transform the original model into a smaller one, and demonstrate the validity of the result by proof or by construction. Indeed, the use of proof checkers for this purpose, in combination with model checkers for the verification proper, has been suggested as an elegant way to combine the strength of these two verification methods. Although this can be useful approach for specific classes of systems, it is less likely to be a generic solution to the problem, as it begs the question. Any generally applicable method seriously risks bringing us right back to the combinatorial explosion that we want to avoid.

In practice, verification models are not formally derived or proved, but constructed on the basis of a combination of insight, heuristics, and sometimes formally well-defined abstractions. This principal loss of a formal link between the formal definition of a system and its effective verification model may be lamented, but it has a positive side to it. The availability of complete formal specifications of systems that we want to verify may help, but is no longer an absolute requirement. This is important as for complex systems such specifications are as a rule not available, and the cost of producing them is often prohibitively expensive. Smaller, more abstract specifications that suffice for the verification of some crucial correctness properties, however, could help to increase confidence in the correctness of a system in a more realistic price-performance ratio.

In addition, it should be realised that the relation between formal specifications of complex systems and their realisations is more problematic than that between programs and their implementations. Complex systems generally cannot be produced by just using reliable compiler(s), and often need elaborate engineering involving both hardware and software, requiring solutions that are unique to the given system. This implies that if we want to assess not only the correctness of a formal design, but actually want to analyse properties of the resulting system, its formal specification may not be the only relevant source of information.

We thus find ourselves in a situation in which the validity of many of our verification models cannot be demonstrated by formal means. This means that if we want to assess their validity, and we must if we take our job seriously, we can only use experimental methods. Moreover, this experimental validation is not just a phase born out of temporary necessity, but it constitutes an essential methodological ingredient for the verification of real-life systems. As in physics, it is the tool to bridge the orders of magnitude that lie between a complete description of a system and an effective theory of its properties (cf., an extremely large set of molecules vs a volume

of gas). It is worthwhile mentioning that in physics one can also quantify the consequences of such abstractions and show that the errors incurred are *sufficiently small*. In this respect it is interesting to note that in *performance analysis* often great simplifications of the evaluation models can be obtained without significant loss of precision. Such approximative abstractions are not feasible if evaluation is restricted to evaluation in the binary system of classical logic. Stochastic interpretations of behaviour can perhaps provide a way forward in this respect: they would allow abstracting away from behaviours that are sufficiently unlikely.[1]

Because the experimental paradigm is foreign to many who are active in the field of formal methods, its role is seldom explicitly addressed, and if acknowledged, it is usually delegated to the engineers of "real" systems and the testing community. But the engineering of verification models is a task that requires intimate knowledge of the formal methods that are used, and therefore should concern all who are interested in the application of verification.

What is urgently needed is agreement on what constitutes good verification practice. Although there is a substantial increase of the number of papers that report on verification experiments, there is no generally acknowledged format for the presentation of the results.[2] It is essentially up to the authors and reviewers of each individual paper to decide what constitutes a scientifically defensible account for the case(s) at hand. It is not uncommon that far-reaching conclusions are drawn on the basis of very limited, or badly presented empirical data.

If we take our inspiration from the established experimental sciences, we should follow a protocol that includes the following ingredients:

- *Problem statement*: clearly defines the problem that is addressed. It answers questions like:
  - What system or design must be verified?
  - What properties must be verified?
  - What assumptions are made?
- *Verification set-up*: describes the ingredients of the verification, their use, and relation accurately, so that all will be *repeatable* by others. Related questions are:
  - What verification model is used?
  - How are the properties formalised?
  - What tools and computing equipment are used? (versions, relevant system parameters)
  - What procedure was followed?
- *Measurements:* gives all the relevant data that were obtained. These include:
  - Verification results of the properties. (verified/falsified/no result)
  - Performance characteristics. (time/memory consumption)
  - Observed irregularities.
- *Error discussion:* evaluates systematically all potential sources of errors that could have influenced the measurements. It is especially this section that is crucial for the interpretation of the results of a verification experiment. In spite of this, most or all of it is missing in many reports. Among its main ingredients are:
  - Quality of the verification model.
  - Accuracy and reliability of the measurement data.
  - Nature of possible errors.
- *Conclusion:* presents the final outcome of the verification. Generally, this cannot be a simple yes/no-answer, but must be a qualitative and/or quantitative interpretation of the measurements, i.e., observed verification results in the light of the error discussion.

Current verification practice is often opaque, not because it does not include activities to validate the verification model, but because it does not make them explicit and does not relate them via an error discussion to the results. What complicates matters is that the debugging of a verification model is often interleaved with the verification itself, when during verification unexpected properties are encountered that are not related to errors in the original system, but to errors in the model. Encountered errors must therefore always be interpreted through careful analysis to separate the two distinct methodological processes: the *validation* of the verification model, and the *verification* of system properties using that model. This can lead to a continuous improvement of the verification model itself, and requires precise bookkeeping of model versions and properties verified. The quality of this process has decisive influence on the quality of the verification procedure as a whole.

So far, we have looked at experimentation as a way to improve the practical applicability of formal verification. However, yet another angle exists that links verification to empirical methods, viz., in the scientific evaluation of formal methods.

4 Verification and the methodology of computer science

From time to time the question concerning the nature of computer science among the sciences is raised. Hartmanis addressed this question in his 1993 Turing Award Lecture, which led to a subsequent discussion published in the *ACM Computing Surveys* [3, 5]. Hartmanis' own conclusion is not very precise: he qualified computer science as a new species among the known sciences for which "a haunting question remains about analogies with the development of physics". Even so, he reports on the coexistence of science and engineering aspects, and remarks: "Somewhat facetiously, but with a grain of truth in it, we

[1] A related plea for a more stochastic view of computer science was recently held by David Tennenhouse [9].

[2] Limited guidance can sometimes be found in handbooks on scientific writing, see e.g., [11].

can say that computer science is the engineering of mathematics (or mathematical processes)". The authors in [3] take various positions, some defending the experimental science point of view, others emphasising the engineering aspects, and yet others argue for both. The overall impression is that (at least at that time) there is no general agreement.

Interestingly enough, already in 1986 Robin Milner gave an account of the experimental nature of a good part of computer science in [7]. He distinguishes between the hard core of computing theory, consisting of recursion and complexity theory, dealing with the characterisation and classification of what is computable, and mathematical theories "in the service of design", i.e., theories that help making and analysing computational artifacts. He proposes that such theories must be evaluated experimentally, by using them in prototype methodologies that are tried out in practice.[3] This position is nicely reminiscent of the point of view taken by Herbert Simon in his *Sciences of the Artificial* [8]. He argues that in general the engineering of artifacts is based on an empirical science of implementation and realisation methods. Experimental design is used to determine the scope of effectiveness of the different methods: under which conditions and circumstances can they be applied successfully, and how do they influence the quality of the resulting product?

In this context (computer-aided) verification can be seen as experiments in the sense of Milner and Simon to determine the effectiveness of formal methods. Of course, there are many other experiments that one can think of, dealing with qualities other than correctness. However, verification is a useful class of experiments because it can be tool supported and seems to lend itself better for purposes of comparison than, for example, entire system designs.

The role of verification as experimentation with formal methods suggests that we should also develop richer evaluation criteria for such experiments than seem to be in current use. Computer-aided verification is strongly focussed on the performance (time, memory usage) of the software tools that are used. This is understandable from the existing drive towards faster and better verification tools. Nevertheless, it is very important to know to what extent other aspects of verification are (not) supported in different formal frameworks, such as the relative ease of validating a verification model, of obtaining a verification model, of selecting and formalising correctness criteria, etc. These observations imply that it is important to repeat and compare verifications using different formalisms and tools. The results of such repeated experiments should be publishable. This should also hold for failed attempts, provided that interesting lessons can be distilled from such failures.

Publications like [6] suggest that computer science compares badly with other branches of science, in the sense that relatively few papers are published with experimentally validated results. This state of affairs emphasises the need for initiatives to rectify the situation, such as the Electronic Tool Integration (ETI) platform of this journal [4]. Every real opportunity to validate our methods should be exploited, and we should strive for a culture that is comparable to that in the other sciences, viz., that in the long run there is no place for formal methods that have not been validated by serious experimentation.

[3] A similar point of view was elaborated by the author in [1]

References

1. Brinksma, E.: What is the method in formal methods? In: Formal Description Techniques, IV, IFIP Transactions C-2. North-Holland, 1992, pp. 33–50
2. Brinksma, E.: Verification is experimentation! In: CONCUR 2000 – Concurrency Theory. LNCS 1877. Berlin, Heidelberg, New York: Springer-Verlag, 2000, pp. 17–24
3. Computing surveys symposium on computational complexity and the nature of computer science. ACM Comput. Surv. 27(1): 5–61, 1995
4. Electronic Tool Integration platform (ETI). `http://www.eti-service.org/`
5. Hartmanis, J.: Turing award lecture: on computational complexity and the nature of computer science. ACM Comput. Surv. 27(1): 7–16, 1995
6. Lukowicz, P., Heinz, E.A., Prechelt, L., Tichy, W.F.: Experimental evaluation in computer science: a quantitative study. J. Syst. Software 28(1): 9–18, 1995
7. Milner, R.: Is computing an experimental science? Technical Report ECS-LFCS-86-1, Laboratory for the Foundations of Computer Science, University of Edinburgh, UK, 1986
8. Simon, H.A.: Sciences of the Artificial. MIT, Boston, Mass., 1981
9. Tennenhouse, D.: Proactive computing. Commun. ACM 43(5): 43–50, 2000
10. Turski, W.: Essay on software engineering at the turn of the century. In: Fundamental Approaches to Software Engineering. LNCS 1783. Berlin, Heidelberg, New York: Springer-Verlag, 2000, pp. 1–20
11. Zobel, J.: Writing for computer science. Berlin, Heidelberg, New York: Springer-Verlag, 1997