A Perspective on Program Verification*

Willem-Paul de Roever

Christian-Albrechts University of Kiel

Abstract. A perspective on program verification is presented from the point of view of a university professor who has been active over a period of 35 years in the development of formal methods and their supporting tools. He has educated until now approx. 25 Ph.D. researchers in those fields and has written two handbooks in the field of program verification, one unifying known techniques for proving data refinement, and the other on compositional verification of concurrent and distributed programs, and communication-closed layers. This essay closes with formulating a grand challenge worthy of modern Europe.

1 Background

Conjecture: It has become a real possibility that Germany's most powerful industrialist, Jürgen Schrempp, heading the largest industry of Germany, DaimlerChrysler, will be fired next year because his company has not spent sufficient attention to improve the reliability of the software of its prime product, Mercedes Benz cars. For, as a consequence of the poor quality of the top range of Mercedes Benz limousines, BMW has now replaced Mercedes Benz as the leading top-range car manufacturer in Germany. And this fact is unpalatable for the main shareholders of DaimlerChrysler (Deutsche Bank, e.g.).[1]

The underlying reason for this fact is that 60% of the current production of Mercedes Benz cars has to be frequently called back because of software failures, the highest percentage of any car manufacturer in the world. And this percentage cannot be changed in, say, a year, the period of time Schrempp has to defend again his industrial strategy to his shareholders (this year his defense took place on April 6, 2005).

This conjecture is at least the second of its kind: The Pentium Bug convinced the top level chip manufacturers that chips should be reliable and bug-free to the extent that any bug occurring after the production phase should be removable, at least to the extent that patches should be applicable circumventing those bugs.

A third fact, not a conjecture, would be that two crashes of a fully loaded Airbus 380 due to software failure in a row would lead to the demise of the European aircraft industry. And one such crash of the Airbus 380 would have

* wpr@informatik.uni-kiel.de

[1] Last minute news: Schrempp leaves DaimlerChrysler end of 2005 (Kieler Nachrichten 29.07.05).

far more serious human and economic consequences than any Pentium-scale bug could ever have caused.

These three examples suffice to establish that reliable software is now a vital component for the survival of at least three modern key industries: Those of cars, high performance chips, and passenger aircraft. When in ten years from now ecological disaster is inevitable, nuclear reactors will be added to this list.

Conclusion: *The production of reliable software is here to stay.*

2 Producing Reliable Commercial Software: A Distant Grand Challenge

Is currently produced software reliable? The answer is an unequivocal NO.

Can reliable software meeting current demands in volume be produced at his very moment? *No!*

Is reliable software needed in most applications? *Neither.*

For, if the product in which this software is embedded allows for rebooting and if software crashes in this product can be remedied without leading to catastrophe, one can live with sof(t)ware which is only on the average reliable ("sof" means failure in dutch). We do so in our daily lives. So much for the production of "Sof"ware.

Is it possible to produce reliable software? The answer is: Only with the greatest of efforts.

Yet the need for reliable commercial software is not only unarguably there, but also the only forever distant *Grand Challenge* which I can think of within our field of computer science.

3 Why this Challenge is Real: Impediments to Overcome

In this section I shall argue that the use of synchronous languages for stating specifications, and the use of any modelchecking or theorem-proving tool or any combination thereof, and of any disciplined use of mathematics, has led to tremendous successes in verifying specialized carefully-tuned domain-specific applications, but does not have the potential required for mass production of reliable software in general in the near future.

3.1 Formulating Correct Specifications

As scientists in the area of formal methods our future task is, therefore, as clear as is the fact that formal methods are not the only tool needed to meet the demands for producing reliable software.

For as difficult it may be to verify large programs, it is at least as difficult to formulate their specifications.

For instance, even when these specifications are written in executable languages, such as the synchronous languages Esterel, Lustre, and Signal, or the

Statemate family of languages, the resulting specifications are programs which should be proved correct, too, relative to new specifications, which should be correct in their turn.

Consequently, there will always be a moment when these specifications should be tested, be it by trail and error, by inspection, or by deducing their logical consequences and testing whether these are acceptable. Famous errors made in the recent past point in this direction. For instance, the fact that one part of the positioning system of a recent marslander was written in the metric system, whereas another part was formulated w.r.t. the English systems of weight and measurement, might have been detected by simply testing it in a realistic simulation, as done, e.g., in trials in the Statemate system.

In one case I know of, the efforts involved in producing correct software have been successful once a correct specification has been found: Once a correct specification in a synchronous language has been produced. For compilers translating this kind of specification languages can be proved correct, as has been done for Signal by a team led by Amir Pnueli [5].

However, as soon as asynchronous properties have to be integrated within a synchronous framework, the race is open, as the field of GALS (for Globally Asynchronous Locally Synchronous) specifications shows, initiated by Albert Benveniste.

3.2 Verifying that a Specification is Satisfied using Modelchecking

Then, given the availability of correct specifications, is it possible to verify their satisfaction?

In case of finite state spaces, and in case of infinite state spaces of a very special nature, this can be done using modelchecking techniques. Their advocates will claim that their development is one straight success story, whose end is not in sight. Be this as may, in the (recent) theses produced at my chair in Kiel and under the guidance of the late Rob Gerth at the Technical University of Eindhoven one aspect struck me as essential:

> The enormous effort required to mold very carefully chosen examples in a shape fit for being checked by an implementation of a newly proposed modelchecking strategy.

This effort was caused because any slight deviation from the example at hand:

1. would be uncheckable using the newly developed technique in general, and/or
2. run up against the limitations of the resources employed in the particular implementation.

This observation is made not to belittle the efforts of the researchers concerned. Far from it, I have the deepest respect and highest admiration for the researchers involved in this challenging and highly successful field!

My observation only serves to formulate the obvious:

> The efforts involved in, and the very serious impediments imposed by, overcoming the limitations of any implementation of any modelchecking technique, applied to any serious example I know of during the existence of this now 25 years old field, are enormous.

Usually, modelchecking is associated with checking the correctness of (synchronous) hardware circuits. Gerard Holzmann has developed the Spin modelchecker for checking (asynchronous) software for Bell-Labs.

He states that one of the reasons for his success is Moore's law, in this case, the fact that hardware speed and size of memory double every 18 months. This enabled him, a.o., to verify a concurrent telephone network server for Bell-Labs, i.e., industrial software.

In a similar vein, Siemens (now Infineon's) modelchecker (which is commercially available), has been reported a few years ago to be able to check the software of one industrial robot in a production line, e.g., for a steel production mill. Verifying the correctness of an entire line required at that moment an approximately 6-months effort due to the need to finetune the assumptions of one robot in order to be logically implied by the accumulated commitments of its predecessors in the line, which had to be done by a team of engineers.

Finally, nowadays the importance of modelchecking is predominantly that of a systematic debugging technique, because of its capability to generate traces leading to the errors found.

3.3 Verifying that a Specification is Satisfied using Theorem Proving

The impediments met during modelchecking can be reformulated, mutatis mutandis, for Theorem Proving.

One proud advocate of a draft of a very interesting paper on inclusive end-to-end verification, which I read a few days ago, observed that his team of verification engineers could now verify one page of code using Hoare style verification checked by the theorem prover Isabelle in one week.

Which proves my point: Similar limitations as exist for model checkers hold for verification based on theorem provers. Enormous advances are being made by the teams of Tobias Nipkow, Wolfgang Paul, John Rushby, Natarajan Shankar, and others. However, any production of software verified by a theorem prover is still a major research effort and has been so ever since theorem proving entered the field of program verification 40 years ago.

Verifying that a specification is satisfied using a combination of modelchecking and theorem-proving techniques has entered the field by the development of the tool Invest of Bensalem, Bouajjani, and Lakhnech, integrated into PVS by Sam Owre, and extended at Saarbrücken by Podelski and his team in the context of AVACS (see below). Although one may expect that the same objections hold as for modelchecking and theorem proving, it is clear that verification engineers having such a powerful tool at their disposal may be able to make considerable progress.

Indicative of the promise of this mixed approach is AVACS [2], a German consortium of universities, research laboratories, and industries, led by Werner Damm, aiming at the automated verification of the embedded software of a high speed train by 2016!

However, clearly in their limit at best semi-automatic tools can result from the combination of modelchecking and theorem-proving techniques, requiring specialist support. This defeats the goal of any large scale production of unspecialized verified software, but makes it possible to verify related families of domain-specific software.

3.4 Verifying a Program by a Disciplined Use of Mathematics

One of the most impressive verification efforts I know of has been undertaken by a team led by Leslie Lamport and resulted in the complete verification of two commercial cache-coherence protocols for Pentium-class processors, produced by AMD.

Clearly a team consisting of top engineers, verification specialists, guided by a person of the stature of Leslie Lamport is able to obtain results which only much later can be emulated using any of the aforementioned techniques. Therefore, cloning Leslie might possibly result in producing correct software for specialized industrial needs. Of course, more than cloning alone would be needed, because two clones of Leslie would be very hard to manage...

I'll come back to this possibility because it symbolizes the partisan endeavor of the extremely gifted individual,[2] who, however gratifying he may be for me as a researcher, contradicts clearly any form of mass production of software, whatsoever.

Yet for extremely complicated algorithms, which are to be implemented in hardware, I see no other realistic solution than the cooperation of very gifted individuals. And this will always be the case, since any time has its own frontier of research.

4 Intermediate Conclusion

The only conclusion that can be formulated thus far is the conjecture that when programs of unrestricted complexity are to be verified, it is in the nature of human beings to find ways of proving these correct, given enough time, an unbounded number of resources, and an unbounded number of gifted people.

The problem is, e.g., that:

1. current chip manufacturers, have now hit the boundary of the number of available specialists and resources, and do not know how to proceed now that more of everyone and everything is not anymore available, and that, e.g.,

[2] A reviewer of this paper remarked that also Bob Kurshan belongs to this list.

2. modelchecking has to beat exponential bounds, which can be done at best in a limited number of restricted cases at the cost of tremendous intellectual achievement.

However, as the French "approche synchrone" in combination with Harel's StateCharts have shown, once some of the greatest minds of our time start working at the top of their capacity in the restricted area of specifying real-time embedded systems not only as an academic exercise but at a scale required for specifying the full complexity of software required for operating nuclear reactors or gigantic aircraft such as the A 380, the complementarity of the formal academic approach with respect to the one based on industrial trial and error is established.

For it has led to an improvement in quality of the produced code by a factor of at least hundred.[3]

5 Considerations Involving Programming Languages

Originally, I started out by stating that our real Grand Challenge is the production of reliable software at an industrial scale.

In which languages is such software usually written?

C Operating system software is often written in C, C++, or another dialect of C. Even lightweight verification tools, such as abstract analysis tools have problems with checking software written in C. Some years ago the crash of one of NASA's Marslanders due to software failure could only be tracked after translating its flight software into Java, and then finding the errors using lightweight verification tools for Java, such as those based on abstract interpretations.

An often-heard obstacle against generating correctness proofs for programs written in C is C's reliance on pointer arithmetic. I do not understand this obstacle. Once correctness is crucial, the implementation of integers on a particular family of processors is known and can be easily incorporated in proof systems. Also, the mathematical analysis of pointers is known since its discovery by America and de Boer approximately 20 years ago [9]. However. I do not know of any practical tools based on these particular observations, which work in most practical cases. Important advances have been made by restricting to specific classes of pointer structures, e.g., using monadic second-order logic as specification language for while programs with pointers and by checking the reformulation of the corresponding verification problem using the MONA tool [7].

Very promising and indicative of the state of the art is the effort of proving correctness of a multiprocessor operating system with paged memory techniques written in a seriously curtailed subset of C, undertaken by a team led by Wolfgang Paul [4].

[3] See, e.g., http://www.esterel-technologies.com/technology/success-stories/overview.html, especially the table regarding the Airbus aircraft.

Java Web software is often written in Java. Although it is in principle possible to prove concurrent multi-threaded Java programs correct, by all practical means and purposes such correctness proofs are unattainable, as remarked in recent work by Erika Abraham c.s. in case of deadlock freedom. (Her thesis [1] contains a sound and relatively complete Hoare-style proof system for a multi-threaded subset of Java.)

Yet enormous advances have been made in building tools for verifying restricted classes of properties for certain subsets of Java. Often these operate on abstract versions of these programs; e.g., the Bandera tool [8].

I have heard from Matt Dwyer that the company-internal Java tools of Microsoft are ahead of the published state of the art. Similarly, it has been remarked by the late Rob Gerth that the company-internal modelcheckers of Intel are ahead of the state of the art.

From the point of view of the production of reliable industrial software these are approximately the most positive remarks one can expect: If the reliability of their programs is of such importance to these companies that their internally produced verification tools are kept secret, this is certainly a sign of maturity of our field.

An altogether different approach is producing High Integrity software is obtained by seriously restricting the use of semantically complicated features in a programming language to such an extent that verification becomes commercially feasible. This is the goal of the SPARK approach, documented by John Barnes [6], focusing on software which is correct by construction.

6 Summarizing the State Of The Art

1. The enormous effort required to mold very carefully chosen examples in a shape fit for being checked by a(n implementation of a newly proposed) model checking strategy, theorem prover, or a combination thereof, is still staggering. However, also helped by the effects of Moore's law, ever more programs of industrial importance are being verified using these tools.
2. Once a correct specification in a synchronous language such as Esterel, Lustre, or Signal has been produced, its automatic translation in C is correct when produced by a verifying compiler for that synchronous language. E.g., for the synchronous language Signal, such a verified compiler is available [5]. The associated strategy of producing graphical specifications, e.g., by using the Scade tool, Esterel Studio, or Statemate, and translating them automatically, has led to an improvement in quality of the code produced for real-time embedded systems by a factor of at least hundred.
3. By seriously restricting the use of semantically complicated features in a programming language, it has become feasible to prove the correctness of large programs such as operating systems. The first time this possibility was seriously pursued was in the SPARK-ADA project [6].
4. For extremely complicated algorithms, which are to be implemented in hardware, I see no other realistic solution for their verification than the cooperation of gifted individuals, led by a person of exceptional stature.

7 Grand Challenge

As grand challenge worthy of modern Europe I propose:

> *Proving the correctness of the safety-critical parts of the software for the Airbus A380.*

Acknowledgments. Dennis Dams suggested many improvements, Martin Steffen and Marcel Kyas helped me with getting the paper ready.

Selected References

1. An Assertional Proof System for Multithreaded Java – Theory and Tool Support, Erika Abraham, Thesis, University of Leiden, `www.informatik.uni-freiburg.de/~eab`, 2005.
2. AVACS, Automatic Verification and Analysis of Complex Systems, `www.avacs.org/`, 2004.
3. Süddeutsche Zeitung, Thursday, 7.04.2005.
4. VERISOFT, `www.verisoft.de/`, 2003.
5. Validating the Translation of an Industrial Optimizing Compiler, I.Gordin, Raya Leviathan, Amir Pnueli, ATVA 2004: 230–247.
6. John Barnes, High Integrity Software: The SPARK Approach to Safety and Security, Addison-Wesley, 2003.
7. Jakob L. Jensen, Michael E. Jørgensen, Michael I. Schwartzbach, Nils Klarlund, Automatic Verification of Pointer Programs using Monadic Second-Order Logic, In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97), pp. 226–234, 1997.
8. John Hatcliff, Matthew B. Dwyer, Corina S. Păsăveanu, Robby, *Foundations of the Bandera abstraction tool*, in: The Essence of Computation, pp. 172–203, Springer-Verlag, 2002.
9. Pierre America, Frank de Boer, Reasoning about dynamically evolving process structures, Formal Aspects of Computing 6(3), pp. 269–316, 1994.