Towards a Worldwide Verification Technology

Wolfgang Paul

Saarland University, Computer Science Dept., 66123 Saarbrücken, Germany
wjp@wjpserver.cs.uni-sb.de

1 Introduction

Verisoft [1] is a large coordinated project funded by the German Federal Government. The mission of the project is i) to develop the technology which permits the pervasive formal verification of entire computer systems consisting of hardware, system software, communication systems and applications ii) to demonstrate in collaboration with industry this technology with several prototypes. During the fall and winter of 02/03 this project was planned by a task force headed by the author.

This task force had to face issues very closely related to what we have discussed in Zurich and we have lived now with the decisions made early in the year 2003 for more than two years. Based on this—mostly positive—experience we make eight scientific, technological and administrative suggestions for the worldwide coordination of efforts in software verification.

2 Basic Research Versus Technology

Basic research identifies fundamental effects and laws. It also develops laboratory prototypes. Laboratory prototypes demonstrate, how newly discovered laws and effects may be applied: *something like a laboratory prototype is expected to work* in engineering. Turning this prototype into a component of a technology is left to the engineers. It requires the elaboration of details which are judged to be boring from a basic research point of view.

A component of a technology must work as it is. And it does not work in isolation. All components of a technology must work together as they are. The world of engineering technology is a binary world; time to work out details is (or at least should be) over: things work or they do not work. Imagine you are dying in an airplane crash due to bad software. That fixing a single line of code would have saved your life is no consolation whatsoever.

In the past, research in the field of verification has stressed the basic research aspect, very much at the expense of the engineering aspect. As a consequence, we now have CAV tools which are numerous, ingenious and often even powerful; because that is appreciated and rewarded in basic research. But i) the landscape of our tools is still very poorly integrated and ii) for many of our tools it is not clear whether they prove correctness with respect to exactly the right specifications.

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 19–25, 2008.

Clearly, something like the specifications we are presently using will eventually permit to prove the correctness of big pieces of software. But then, it is also very likely that something like the original software would work in the first place.

3 Right Specifications and Stacks

A specification can be bad for two reasons:

i) It does not capture the user's intention. In general this cannot be discovered by mathematical methods alone. ii) In a computer system with layers, it does not permit to deduce desired properties of the next layer upwards or it cannot be proven using properties from the layer below.

Fortunately there are no principal difficulties to test, whether a specification works together with correctness theorems of other system layers. One simply tries it. That it can be done has been demonstrated as early as 1989 in the famous CLI stack [2]. Experiences since 2003 with far more complex stacks in the Verisoft project are also very encouraging.

We therefore judge it necessary, that the development of a technology for software verification has to be carried out in the context of the verification of entire stacks.

A case study: compilers in stacks. We inspect three well known sources of definitions of programming language semantics: i) the classical Hoare/Wirth paper on Pascal semantics [3] ii) the textbook of Nielson and Nielson [4] iii) the textbook of Winskel [5].

In [4,5] variables can range over the natural numbers. For a program running on a finite processor there is no way to prove this. In contrast, *int* is a finite data type in [3]!

Real programs run under operating systems and they perform I/O. Their computations are *interleaved* with the computations of other programs. Modelling this requires small steps semantics. Thus one cannot rely exclusively on the (big steps) definitions in [3].

Nevertheless the definitions of [3] are a component of engineering technology. Because theorems proven in Hoare logics hold in the corresponding small steps semantics (for the proofs see e.g. [4,5]) they can be used exactly as they are to prove properties of terminating portions of programs. Experience from the Verisoft project suggests that this should by all means be done: productivity with Hoare logics is *much* higher than with small steps semantics alone.

If the verification of an operating system is also part of a project, then conventional small steps semantics alone does not suffice either: one needs to consider in line assembler code for the following reasons: i) arguing in high level language alone is impossible: an operating system written in high level language alone could see in its own variables neither the processor registers nor the user processes. ii) arguing on the assembler language level alone (as was done in [2]) would not be productive enough.

4 Paper and Pencil Theory

Let us assume that we succeed to formally verify a complex stack. Then we can *necessarily* produce a human readable transcript of the formal correctness proof. This proof would be part of a big unified theory of computer science which would i) be (at least!) as stringent as the classical mathematical theories, ii) include in a unified way big parts of what is today called theoretical computer science, and iii) have real systems as examples.

We believe that progress will be faster if this theory is developed first with paper and pencil. In the language of G. Hotz these paper and pencil proofs then can serve as building plans for the formal proofs.

A case study: from gates to user processes i) Hardware is easily specified in the language of switching theory. ii) The random access machines of theoretical computer science are appropriate for specifying instruction sets. iii) Small steps semantics of high level languages is specified by abstract interpreters. iv) various models for distributed computation permit to treat communicating user programs.

Clearly in a verified stack one needs simulation theorems between different layers. Processor correctness is between models i) and ii). Compiler correctness is between models ii) and iii). Operating system correctness—with the scheduler abstracted away—is (because of the in line assembler code) between models ii) and iv). In the Verisoft project the paper and pencil proof for operating system correctness required the introduction of two more parallel models of computation between models ii) and iv): one for operating system kernels and one for operating systems without abstracting away the scheduler.

5 Standardizing Language and Tools

Worldwide cooperative effort is impossible without establishing a common language. In software engineering there is a small number of standard programming languages, among them C and Java. The semantics are admittedly not too well defined, but compilers of a small number of large vendors establish a small number of de facto standards.

For the establishment of a worldwide verification technology we need as a counter part a small number of standard CAV systems. They should be

1. cheap mass products maintained by companies
2. universal: formalization of arbitrary mathematical statements and arguments should be reasonably straight forward. Presently interactive high order logic provers seem the best candidates.
3. easily extendible by automatic tools; interfaces to do this must be open. Without such interfaces we close the door to continuous increase in productivity.

Only with compatible standard tools can different groups of engineers exchange or trade (!) proofs. In the Verisoft project the standard tool is Isabelle/HOL [6].

6 Standardizing Definitions

Some crucial formal definitions should be standardized and maintained (in standard language) in downloadable form on certain web sites. Examples are i) the semantics of some standard instruction sets (note that this includes the IEEE floating point standard). ii) small steps semantics and Hoare logics for C and Java iii) the semantics of certain standard real time operating systems (such standard systems exist for instance in the automotive industry).

7 Establishing Repositories of Verified Standard Components

For all standard components of computer systems both i) verified constructions and ii) their formal correctness proofs need to be made available in repositories. Clearly, for an *industry* of computer system verification, formally verified components from the following list are indispensable: i) processor with optional memory management units and I/O devices, ii) assembler and linker, iii) optimizing compiler for object oriented languages, iv) operating system kernel, v) operating system, vi) distributed real time operating system, vii) interface compiler supporting port mapper and client-server RPC mechanisms, viii) TCP/IP, ix) mail server, x) electronic signature server, and xi) several cryptographic protocols.

Except for compiler optimization and object orientation all items of the above list are milestones of the current Verisoft project. Whenever possible, constructions and arguments from established textbooks were taken as a starting point for the development of the system components and the correctness proofs.

8 Rewarding Engineering Work

Even the best technical decisions are useless if they are not supported by project members. Students will be reluctant to do work which will not lead to a thesis. Researchers at universities will be reluctant to do work which they cannot publish. We must therefore give proper rewards for the engineering aspect of our work. In particular we have to establish a forum for the following kind of results:

1. The integration of known automatic methods in known interactive provers if that increases the productivity on large realistic benchmarks. This in turn requires the publication of such benchmarks *in a form which is easy enough to read*; putting existing large formal proofs on the web is not enough. In the Verisoft project we are working on such benchmarks together with the Southern Methodist University at Dallas.
2. The 'mechanization' of existing paper and pencil correctness proofs for major system components in a CAV system, if similar proofs were not mechanized before. In an engineering sense one can simply not trust paper and pencil proofs alone; in this respect we agree with [7]. In contrast a mechanized proof establishes not only trust in the verified component. It also shows, that the

line of arguments that is used is complete and hence should work in future similar proofs.

Failing to reward the engineering aspects of our work as highly as the basic research aspects will slow down the development of technology.

9 Summary of Suggestions

In order to help establish a worldwide technology of software verification the author suggests to

1. clearly recognize the difference between basic research and engineering,
2. to study stacks in order to get specifications right,
3. produce paper and pencil proofs first,
4. recognize the need for a grand unified theory of computer science,
5. agree on a small number of standard tools permitting both interactive and automatic work,
6. standardize key definitions,
7. make formally verified standard constructions available in repositories and
8. establish a proper reward structure for results with a strong engineering flavor.

In Verisoft, the standard processor is a DLX machine with memory management units [8]. Formal verification in PVS is complete; proofs are presently being ported to Isabelle/HOL. The standard language is $C0$; in a nutshell this is Pascal with C syntax. CO_A is $C0$ with in line assembler code. We are using small steps semantics for $C0$ and $C0_A$. A Hoare logic for $C0$ is used to increase productivity [9]. Formal verification of a non optimizing $C0_A$-compiler is expected to be completed in early 2006 [10]. Formal verification of an operating system kernel written in $C0_A$ is expected in 2006 [11]. Paper and pencil theory for processors with I/O devices and real time systems can be found in [12] and [13].

References

1. The Verisoft Consortium: The Verisoft Project, `http://www.verisoft.de/`
2. Bevier, W.R., Hunt Jr., W.A., Moore, J.S., Young, W.D.: An approach to systems verification. J. Autom. Reason. 5(4), 411–428 (1989)
3. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Inf. 2, 335–355 (1973)
4. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley, Chichester, 1992, revised online version: 1999
5. Winskel, G.: The formal semantics of programming languages. The MIT Press, Cambridge (1993)
6. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
7. Millo, R.A.D., Lipton, R.J., Perlis, A.J.: Social processes and proofs of theorems and programs. Commun. ACM 22(5), 271–280 (1979)

8. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 301–316. Springer, Heidelberg (2005)
9. Schirmer, N.: A verification environment for sequential imperative programs in Isabelle/HOL. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 398–414. Springer, Heidelberg (2005)
10. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: Aichernig, B., Beckert, B. (eds.) 3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany, pp. 2–11 (September 5-9, 2005)
11. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005)
12. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD 2005, pp. 309–316. IEEE Computer Society, Los Alamitos (2005)
13. Beyer, S., Böhm, P., Gerke, M., Hillebrand, M., In der Rieden, T., Knapp, S., Leinenbach, D., Paul, W.J.: Towards the formal verification of lower system layers in automotive systems. In: ICCD 2005, pp. 317–324. IEEE Computer Society, Los Alamitos (2005)

A Discussion on Wolfgang Paul's Presentation

Greg Nelson

Wolfgang, I have one question about the C0 language that you defined. Does it have a storage deallocation function like free()?

Wolfgang Paul

No, it has a new()-operation. And we have a paper-and-pencil proof for a garbage collector. And we are optimistic that it takes about one Russian person-year to verify that thing using the tools we have.

Greg Nelson: Thank you.

Willem-Paul de Roever

A minor question. You did not say in your slides what is the duration of this project. We know that it is three and a half million ECU per year, but is it six or eight years?

Wolfgang Paul

Well, the people who have urged me to run the project are planning for eight years. Now the formula: We are now in the third year. Now, if I get the financing for four-year periods at a row, I lose all power. So, financing is always every two

years, okay? Only for two years we do apply for the next money, because otherwise, I cannot convince some of my collaborators to stay within the direction of the entire project.

Willem-Paul de Roever: It's German practice.

Wolfgang Paul

No, I think they would have given me the money in three-year periods. But it is better to reevaluate after two years and say: "You don't agree anymore with this. Do your research elsewhere, and we spend the money in a different way."

Ramesh Bharadwaj

In light of what Amir said, you seem to have missed the word "specification", and you have not explained exactly what is specified and what is not...

Wolfgang Paul (*interrupts)*

It is usually an operational semantics of an operation. So, for instance, [for an] operating system kernel, I am giving you the operational semantics. I am showing you the user-visible data structures of an operating system kernel, I am showing you the operations, and I am defining the effect of the user-visible data structures. So, in these system correctness proofs, most of these proofs are simulation theory, showing that the lower layer of the system simulates the upper layer of the system.

Ramesh Bharadwaj: OK, thanks.