# The Verified Software Challenge: A Call for a Holistic Approach to Reliability

Thomas Ball

Microsoft Research, Redmond, WA, USA, Email: tball@microsoft.com

## 1 Introduction

The software analysis community has made a lot of progress in creating software tools for detecting defects and performing proofs of shallow properties of programs. We are witnessing the birth of a virtuous cycle between software tools and their consumers and I, for one, am very excited about this. We understand much better how to engineer program analyses to scale to large code bases and deal with the difficult problem of false errors and reducing their number. We understand better the tradeoffs in sound vs. unsound analyses. The software tools developed and applied over the last eight years have had impact. This list of tools includes Blast [HJMS02], CCured [NMW02], CQual [FTA02], ESC/Java [FLL+02], ESP [DLS02], Feaver [Hol00], MAGIC [CCG+04], MC [HCXE02], MOPS [CDW04], Prefast [LBD+04], Prefix [BPS00], SLAM [BR01], Splint [EL02] and Verisoft [God97], to name a few.

This *bottom-up* approach to improving code quality will continue to be successful because it deals with a concrete artifact (programs) that people produce, has great economic impact and longevity. Furthermore, because many of the tools listed above are specification-based, they are easy to extend to new classes of bugs. Finally, a lot of the science to support the development of these tools has been done; there is now before us a long road of engineering to make these tools truly useful and useable by a wide audience.

To balance these great efforts, we should devote some attention to *top-down* approaches to building reliable software and reconsider how we design software systems so that they are reliable by construction. We want to construct software in such a way that, as a result of its structure, it has certain good behaviors and does not have other bad behaviors. Furthermore, these good behaviors (or the absence of the bad ones) should be preserved as we refine the system to add functionality or improve performance.

I am concerned that Verified Software focuses on the analysis of artifacts (programs) rather than on their design and construction. Instead of just positioning ourselves as software critics ("thumbs-up" or "thumbs-down"), we also must get up on stage and demonstrate methods for designing and building reliable software. A holistic approach to the problem of software reliability that encompasses designing, building and analyzing software systems will offer the greatest challenges and rewards.

## 2 Perspective: The SLAM and SDV projects

What is SLAM about? SLAM was motivated by the program of Windows device drivers. What is a device driver? An idealized view of device drivers is that

they are programs that interface hardware to the operating system (the reality in Windows is much more complex). The problem with programming device drivers is that the driver API (essentially a subset of the Windows kernel) is very complex and difficult to master and most device drivers are written by third-party device manufacturers, who are not Windows kernel experts. The device driver "monster" is alive and continues to wreak havoc: device driver failures are the source of 85% of Windows XP crashes.

Why is programming a driver so complex? First and foremost, the Windows kernel is a highly concurrent system and this concurrency is exposed to the driver programmer. Asynchronous I/O is a core facet of the Windows kernel underlying the driver API; it allows for high-performance implementations but is hard to program correctly. The API provides many ways to synchronize access to data and resources. Other major features that the driver API supports are Plug-and-Play (the ability to remove devices and OS features while the computer is on and have everything work correctly) and power management (the ability to selectively power-down subsystems, hibernate and restore power as needed). Like many APIs, the driver API evolved in a very demanding and constrained environment where performance and backward compatibility were driving forces. These forces resulted in a complex API. As a result, driver reliability suffered.

The key idea of the SLAM project is that while a device driver contains a huge amount of state, we need only reason about a small amount of this state when checking whether or not a driver properly uses the driver API. The SLAM tool reverse engineers a Boolean program (pushdown automaton) from the C code of a driver that represents how the C code uses the driver API. Bits in the Boolean program represent important observations about the state of the C program (with respect to its usage of the driver API). SLAM then applies symbolic model checking on this Boolean program. The SLAM engine has been packaged up in a tool called Static Driver Verifier (SDV) that has been released in beta form this year to third party driver developers.

The SDV tool is much more than the SLAM engine. The challenge of making an effective "push button" tool was much more than the creation of SLAM's software model checking engine: the real challenge was the creation and refinement of a set of API usage rules that encode the proper usage of the driver API, as well as an environment model to represent the Windows kernel. Because the driver API is so complex, the development of these rules and environment took a very long time. At first, the rules and environment were too simple and resulted in many false errors. Only after much iteration with driver experts, did we end up with a set of rules and an environment that was effective. During this time, the SLAM engine changed very little. Most of the false errors were due to problems with the rules and environment. This effort was deemed to be worthwhile because we have captured a huge amount of domain expertise about device drivers that now can be leveraged in an automated tool that many driver developers will use. In this sense, SDV is a tireless and expert device driver code inspector that will help driver developers to make proper use of the driver API.

## 3 Will SLAM and SDV Solve the Driver Problem?

The answer is "probably not". Let's try and understand why.

The SLAM project often is held up as a model of success for software analysis. It is nice to receive such kudos but let's not get carried away here! Once, I was invited to give a talk about SLAM at a workshop and saw that in the preliminary publicity materials for the event, the organizer said something to the effect that "SLAM made a significant contribution to improving the reliability of the Windows operating system." I nearly fell out of my chair when I read that, knowing that all the efforts that people in the Windows organization had made over many years to improve Window's reliability and knowing that SLAM's contribution, even if "significant", would pale in comparison to these efforts. (I very quickly got the message removed from the publicity materials, by the way). In reality, it will be years before we can tell what the effect of SLAM on the reliability of Windows will be and it is even questionable, given all the other methods in play for increasing the reliability of Windows, whether we will be able to separate out SLAM's effect.

There are many reasons device drivers fail and SLAM is capable of finding only a certain class of errors. For example, SLAM does not check that a device driver does not write outside the bounds of the data structures it manipulates. A device driver that corrupts the state of the kernel will not be detected by SLAM. SLAM checks the code of a device driver without considering the interleaving of other threads of execution, so it cannot find deep concurrency errors. It does not do performance testing of the driver under heavy volumes or check for reactivity or real-time constraints.

While SLAM is an important tool in the driver reliability toolbox, it is not sufficient to guarantee the reliability of device drivers. There are many other approaches to dealing with misbehaving drivers. For example, some have proposed running the driver in a sandbox where it cannot damage kernel data structures. Now, some may look at this situation and say "so we need more tools". But others may say "we need to design a better APIs for programming drivers". Others may say "we need to redesign the operating system".

My main point is that tools like SLAM and SDV come in very late in the software production process and, as a result, can only have a limited affect on reliability. The tools only are applied after many important decisions have been made (driver API designed, drivers written in C, drivers run in the same address space as the kernel, etc.) that affect driver reliability in ways that are hard for static analysis tools to address.

## 4 Provocation

A program is a very detailed solution to a much more abstract problem. Leading from a problem to a program is a complex process. By focusing too much attention on the program, we risk ignoring the complicated process that starts with a problem and ends in a program. It is this process that, in the end, is primarily responsible for the quality of the program. The process of design matters greatly, with particular attention to the fact that programs are deeply integrated into

our physical world. As is well-known, a critical design mistake made early in this process, once deeply woven into the intricate software tapestry of a program, is not easily corrected.

Viewed from this perspective, program analyzers, model checkers, verifying compilers and tools like SLAM come into play long after a lot of the important work has been done. These tools find errors and suggest minor modifications to the product. Of course, because of the nature of software, these minor modifications can rid the software of crippling behavioral problems. But, in the end, the analysis tools are working at such a low-level of abstraction (the code) that they cannot see the forest for the trees.

An analogy to automatic parallelization was suggested to me by Jim Larus. For twenty years, researchers worked on techniques for automatically parallelizing sequential FORTRAN programs. While this research produced some interesting ideas, it never succeeded in creating efficient parallel programs from sequential ones, mainly because there wasn't a lot of parallelism to be found in these programs in the first place! Today, the parallel program community recognizes that to get efficient parallel programs you need to carefully design parallel algorithms rather than hoping that a tool will be able to extract parallelism from a program that has little of it to offer.

Similarly, we cannot expect verification tools to inject high reliability into a program that was not designed with reliability in mind from the beginning. We must think about reliability at every point in the software production process. If the starting point for verification is that we are given a program and must attempt to verify it, we are in a losing position because we have so little leverage to affect the design of that program. Starting at the "bottom" means our potential energy and potential for success is very low.

## 5 A Call to Action: Software Design Methodologies

I believe that by designing, building and analyzing software in new ways that substantially increases its reliability, we will face many more challenges and opportunities than if we limit ourselves to analyzing software. This is what I mean by taking a "holistic" approach to issues of software reliability. As an example, at Microsoft Research there is a new operating system research project called Singularity led by Galen Hunt and Jim Larus. Here is part of the "Motivation" section from the Singularity Design Motivation [HL04]:

> Singularity is a cross-discipline research project focused on the construction of dependable systems through innovation in the areas of systems, languages, and tools. We are building a new research operating system, called Singularity, as a laboratory for designing systems, extending programming languages, and developing new techniques and tools for specifying and verifying program behavior.
>
> Singularity is the first OS to enable anticipatory statements about system configuration and behavior. A specific Singularity system is a self-describing artifact, not just a collection of bits accumulated with at best an anecdotal history. Singularity's self description includes specifications of the components of the system, their behavior, and their interactions. One can, for example,

> examine an offline Singularity system image and make strong statements about its features, components, composition, and compatibility.
>
> The Singularity research team defines operating system research as research into the base abstractions for computing and research into implementations of those abstractions as exposed by the OS. By returning to this basic definition of OS research, Singularity embraces the opportunity to re-think OS abstractions and their implementations.
>
> OS research is ready for a revolution. Modern systems are bound by abstractions largely defined in the early 1970s. OS research has not kept pace with changes in application composition or security needs of everyday usage scenarios.

Now, you can go read the rest of the note to get some more detail about this project. But just reading this motivation gets me excited. This sounds like a challenge problem! Why? This project has several important attributes that catch my attention:

- it focuses on an important domain (operating systems) with high commercial and societal impact;
- it recognizes that reliability depends on many puzzle pieces, such as operating systems design, new languages and software tools (it is cross-disciplinary);
- it recognizes that the search for appropriate abstractions is a key problem in operating systems design.

There is something to learn from Singularity, even if operating systems is not one's cup of tea. First, let's start with a problem domain (or two or three, to keep everyone interested) for which high reliability is a necessity and work our way towards a solution. (To reiterate, software is a solution to a problem. If we start with the solution then we have no control over how the solution came into being.) We may sacrifice claims to generality but we will gain a lot more (especially, credibility). Second, by necessity, we need to form cross-discipline teams. We cannot expect to acquire all the domain knowledge ourselves. By bringing our expertise in what is possible to specify and verify together with the knowledge of domain experts, we can make much more progress than if we work in isolation. Third, let's look into a variety of approaches to ensuring reliability. We have some specification languages and modeling languages, but are they really sufficient for the domains we wish to tackle? We have model checkers, theorem provers and program analysis but are they up to the task? We should let the problem domain guide our search to the abstractions, methods, languages and tools that will be the most appropriate to that domain, rather than letting the technologies we know and love blind us to other possibilities.

## 6 Conclusion

Based on my experience with the SLAM project, I would like to spend more time thinking about how to design and build reliable software rather than analyzing it after the fact. I don't have a recipe for success or prescriptive advice at this point. However, I am intrigued by the continued success of design patterns and reusable components in helping to construct larger and more complex

systems. For example, Microsoft's `www.gotdotnet.org` site has a community of over 5,000 developers who contribute to the Enterprise Library, "a collection of reusable components that help you quickly build better applications with more features and *higher quality*" (my emphasis). As with the Singularity project, these developers are engaged in the search for better abstractions. We should think about partnering with such people—they are involved in designing and building real applications and have substantial domain expertise. What they are missing is the knowledge we have about verification technology.

## References


[BPS00] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057, pages 103–122. Springer-Verlag, 2001.

[CCG+04] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *TSE: Transactions on Software Engineering*, 30(6):388–402, 2004.

[CDW04] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *NDSS: Network and Distributed System Security Symposium*, pages 171–185, 2004.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI 02: Programming language design and implementation*, pages 57–68. ACM, 2002.

[EL02] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[FTA02] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming language design and implementation*, pages 1–12. ACM, 2002.

[God97] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186. ACM, 1997.

[HCXE02] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI 02: Programming Language Design and Implementation*, pages 69–82. ACM, 2002.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, January 2002.

[HL04] G. C. Hunt and J. R. Larus. Singularity design motivation. Technical Report MSR-TR-2004-105, Microsoft Research, December 2004.

[Hol00] G.J. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.

[LBD+04] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[NMW02] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL '02*, pages 128–139. ACM, January 2002.