Journal of Universal Computer Science, vol. 13, no. 5 (2007), 602-606
submitted: 7/5/07, accepted: 25/5/07, appeared: 28/5/07 © J.UCS

Software Is More Than Code

Sriram K. Rajamani
(Microsoft Research, Bangalore, India
sriram@microsoft.com)

Abstract: This paper reviews the current practice of software engineering and outlines some prospects for developing a more holistic and formally grounded approach.
Key Words: Formal Methods, Software Engineering
Category: D.2.4, F.3.1

1 Introduction

Building large scale software with both high quality and effectiveness is a huge challenge. Quality is meant in the sense that the software should be robust, reliable, secure, and serve the intended needs of users, and effectiveness is meant in the sense that the group of people building the software should be productive, and effective at their tasks. The process of building large scale software contains several stages including:

• Requirements gathering: understanding what software needs to be built and how the software is intended to interact with its environment. The environment typically includes users, hardware and other software.

• Design: coming up with the architecture of the software that meets the requirements, taking into account constraints on resources and performance.

• Implementation: coding the design in a programming language, and fleshing out all the details, so as to conform to the design, and satisfy all the requirements.

• Testing: checking if the software indeed satisfies requirements by trying it out on important scenarios, whether it functions with expected performance, whether it is robust, reliable and secure in terms of adversarial inputs.

• Deployment: installing the software to the users, and collecting feedback on how the software works.

These stages are typically iterated in the software life cycle. Most successful software has a long shelf-life, and is typically modified and evolved so as to fix errors, and accommodate changing requirements. This article surveys the state of the art in tools and methodologies for building software, and points out opportunities for further research.

2 State of the art

Several people with varying backgrounds and skills need to interact to produce useful software. Gathering of requirements is best done by domain experts that understand the intended use of the software. For example, insurance experts are the best people to determine the requirements of insurance software. Design is a black-art that is not well understood today. People get 'good' at design, from the experience of repeatedly building software. Implementation, testing and deployment are done by programmers who typically have basic education in computer science, or have basic education in mathematics or science or engineering with some training in programming. From requirements to deployment, the 'intent' of some domain experts gradually gets refined to 'working code'.

2.1 Tools for implementation and testing

Over the past decade or so, we have witnessed a revolution in tools to improve productivity of implementation and testing. The biggest breakthrough in this area has been the use of static analysis. Static analysis works by examining the program without running it, and can check if the program could ever violate simple properties. This is fundamentally different from running the program on a few scenarios during testing, and ensuring that violations don't occur during those runs. In contrast to testing which only finds errors that 'did happen', static analysis finds errors that 'could happen'. The flip side of static analysis is that some of the errors reported by static analysis tools might be 'false errors' that can never manifest in any concrete execution of the program. Static analysis tools are intended to augment, rather than replace, testing. These tools do not typically ensure that the software implements intended functionality correctly. Instead, they look for specific kinds of error more thoroughly inside the program by analyzing how control and data flow through the program.

Heuristic analyzers. Heuristic analyzers such as PREFix [Bush et al. (2000), Larus et al. (2004)], PREFast [Larus et al. (2004)] and Metal [Engler et al. (2000)] do not attempt to cover all paths. Further, along each path they do approximations. However, they manage to exercise code paths that are difficult to exercise using testing. Thus they are able to detect property violations that remain undetected after testing. Due to their heuristic nature, they are neither sound nor complete. They manage false errors by using filtering mechanisms to separate high-quality error reports, and statistical techniques to rank error reports. However, these tools have provided impressive utility to their users. PREFix and PREFast have been successful in reporting useful errors over tens of millions of lines of Windows code, and are now used routinely as part of the Windows build process. Metal has similarly found useful errors over several millions of lines of open source code.

Sound analyzers. Sound analyzers explore the property state machine using a conservative abstraction of the program. Usually, the abstraction used is the control flow graph, augmented with the state machine representing the property. Thus, the analyses explores all the feasible executions of the program, and several more infeasible executions. However the analyses do not explore individual paths. Instead, they explore abstract states.

The complexity of the analysis is typically the product of the number of nodes of the property state machine and the size of the control flow graph of the program. Thus, for a 100,000 line program, and a 5-state property, the analysis can be done in 500,000 steps which is very feasible on modern processors. However, sound analyzers are necessarily incomplete, and consequently report false errors. A promising technique to reduce false errors is counterexample driven refinement [Kurshan (1994), Clarke et al. (2000), Ball and Rajamani (2001)]. Here, abstract counterexamples are simulated in the original program to check if they are true errors. If they are not true errors, then the analysis automatically adds more state to track in the abstraction. Counterexample driven refinement has been used to build tools that have a very low false error rate [Ball and Rajamani (2001), Henzinger et al. (2002), Chakiet et al. (2004)]. Expressive type systems have also been used to state and check properties [DeLine and Fähndrich (2001), Foster et al. (2002)]. Since types are integrated into the programming language, the approach has several advantages. Recent approaches allow enhanced programmability of properties using types [Chin et al. (2005)]. While type based approaches are very natural for specifying protocols on one object at a time, they have difficulties specifying protocols that involve multiple objects. Abstract interpretation [Cousot and Cousot (1977)] is a generic theory for building sound static analyzers. Tools based on abstract interpretation have been tuned using domain knowledge to produce very few false errors in large safety critical software [Blanchet et al. (2003)].

Early detection of programming errors using static analysis has come of age in the past decade, and is widely used in industrial practice in companies like Microsoft [Larus et al. (2004), Hackett et al. (2006), Ball et al. (2006)].

3 Opportunities for improvement

In spite of the advancements in implementation tools, the biggest problem in software engineering continues to be the bridging of the 'gap' between the intent captured in requirements and expressed at a high level, and the detailed encoding of this intent in the code. There are no good tools, either mental or mechanical, that allow comprehension of large programs, and provide a mapping between how different parts of the code work together to satisfy the requirements. Thus, looking for any high-level requirements within a million line program is analogous to running around New York City looking for a lost cat, without a map, and without any street signs. This problem cannot be remedied without a fundamental change in the way software is built, and without changing our belief about what software is.

So, what constitutes software? Today, most people would say that software consists of code written by programmers, and the compiled executable that runs on the hardware, and is shipped to customers. Herein lies the problem. Software needs to be much more than an instruction stream that communicates with the hardware at a detailed level about what instructions to execute. It needs to be a medium of communication between all the different people who are using it and building it, ranging from the users, domain experts, architects, developers and testers. It needs to be a repository of actual requirements that the software is intended to satisfy, high level design decisions that have been taken about the architecture of the code, and how different components of the software interact through interfaces. Such information is present only in the 'brains' of senior developers and architects in software companies today. Much can be improved if this information can be represented as higher level abstractions of the software, and if these higher level abstractions can be maintained and kept synchronized with code, as the code evolves.

This would require innovations in the way requirements are specified, the ways by which architectural and design decisions can be represented, rich description of interfaces that convey more of the semantics of the interfaces, and tools and technologies to tie these down to the actual code. Such knowledge has to be imparted as part of education given to both software practitioners and computer scientists. This would require people with different skills, domain experts, programmers, testers, and academic computer scientists to communicate and work together. Most importantly, it requires a fundamental change in attitude as to what constitutes software. If that attitude is changed, we can certainly hope that someday we will consummate David Parnas' dream: that software engineering becomes a true marriage between computer science and engineering [Parnas (1997)]!

References

[Ball et al. (2006)] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, S. K. Rajamani B. Ondrusek, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys 2006*, pages 73–85. ACM Press, 2006.

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

[Blanchet et al. (2003)] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI 03: Programming Language Design and Implementation*, pages 196–207, 2003.

[Bush et al. (2000)] 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.

[Chakiet et al. (2004)] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[Chin et al. (2005)] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI 05: Programming Language Design and Implementation*, pages 85–95. ACM, 2005.

[Clarke et al. (2000)] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

[Cousot and Cousot (1977)] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[DeLine and Fähndrich (2001)] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*. ACM, 2001.

[Engler et al. (2000)] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.

[Foster et al. (2002)] 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.

[Hackett et al. (2006)] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE 06: International Conference on Software Engineering*, pages 232–241, 2006.

[Henzinger et al. (2002)] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, January 2002.

[Kurshan (1994)] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[Larus et al. (2004)] 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.

[Parnas (1997)] D. L. Parnas. Software engineering: An unconsummated marriage. *Commun. ACM*, 40(9):128, 1997.