Autom Softw Eng (2010) 17: 213–215
DOI 10.1007/s10515-010-0071-y

Editorial: software defect detection

Robert J. Hall

Received: 16 May 2010 / Accepted: 17 May 2010 / Published online: 26 May 2010

The most difficult aspect of producing software is getting it right. Defects plague all software projects and enormous expense and time go into ridding code of the worst of them. Consequently, the ASE Community has devoted much attention to the problem of *software defect avoidance*. This effort can be broken down into *defect prevention* and *defect detection (and removal)*.

- The defect prevention school of research includes both advanced requirements engineering (RE) methods and software synthesis. In RE, the focus is to avoid defects derived from misunderstanding what the program is supposed to do in the first place. In synthesis sophisticated tools derive code from high level specification models, with the assumptions being both that the specs are so high level and transparent as to be easy for domain experts to get right, and that the tools' derivations are validated once and for all, leading to code that is correct by construction.
- The detection & removal school of research starts by assuming that code is produced using existing state-of-the-practice techniques and then sophisticated tools are brought to bear on these artifacts to detect defects that may have arisen during design and coding. Once detected, the defects can be removed either manually or with the help of additional bug localization and diagnosis tools.

Of course, a modern software development shop will use tools and ideas drawn from the fruits of both of these areas to get their products out the door. For example, advanced requirements engineering methods are used in coordination with a mix of code generated by tools from high level models and code produced using traditional techniques. Model defects can be detected by tools like model checkers and theorem

R.J. Hall (✉)
AT&T Labs Research, Florham Park, NJ 07932, USA
e-mail: Bob.ASEJ@gmail.com

provers. These steps are then followed up with testing phases powered by automated test generators.

Both the prevention and detection schools of thought have led to interesting concepts, engineering advances, and useful tools; both are still pursued vigorously today; and both are central to the scope of the ASE Community and this Journal. However, the theme of this issue of ASEJ is software defect detection. All four articles are solidly in this camp, though having widely different archival contributions.

- In testing, the most critical resource is time. The more tests that can be run within the strongly limited time allotted, the more software defects can be detected. Consequently, for systems whose controllers or models are finite state machines (FSMs) at heart, it is advantageous to move smoothly from test to test with as little delay as possible; however, if the order in which states are visited is not planned carefully, the operation of moving between tests can involve many reset operations. Such operations, akin to rebooting a computer, involve lots of non-productive time. In "Generating a checking sequence with a minimum number of reset transitions", Hierons and Ural describe a technique for generating a checking sequence for an FSM that minimizes the number of reset transitions required.
- Software product lines are usually organized by features. Each instance of the product line comprises a particular collection of some of the features, while others are left out. In large product lines, the number of possible instances can be enormous, so the problem of detecting defects in all possible realizable instances is correspondingly daunting. In "Type safety for feature-oriented product lines", Apel, Kastner, Grosslinger, and Lengauer define a type system for such product lines that allows a tool to check the entire code base of the product line once. The type checker detects and flags software defects at compile time. Once the code is repaired to the point the check passes, then all instances are guaranteed to be type safe when realized, thereby guaranteeing the absence of defects with respect to properties expressible in the type system.
- In "SUDS: An infrastructure for creating dynamic software defect detection tools", Larson has produced a framework allowing the software engineer to build and integrate many different defect detection tools, some static (operating on source) and some dynamic (using information found during executions), most of which have been described individually elsewhere in the literature. Larson further shows how these tools complement each other and provide advantages when used together; for example, showing how static analyses can be used to eliminate unnecessary instrumentation checks thereby saving time.
- The article by Edwards, Tucker, and Demsky contributes what promises to be both interesting and useful to the software engineering research community and (by extension) to the beneficiaries of that community. In "AFID: An automated approach to collecting software faults", they describe an automated approach to collecting and archiving case studies of faulty software. This repository, which can in principle grow daily, should provide rich fodder for researchers and practitioners who wish to compare software tools, particularly those aimed

at finding defects. Of course, this does not solve the larger problem of companies being unwilling to release source code and related information to the public, but (a) companies can build such repositories in-house and use them to tune their own tools and processes, and (b) the open source community can build a (presumably quite large) such repository open to university and other researchers.

Please enjoy these solid contributions to the field of software defect detection, and feel free to email me (or the authors) with your thoughts or reactions.