CORRECT-BY-DESIGN SOFTWARE IS FUNDAMENTAL TO HIGH-CONFIDENCE DEVICES

S. J. PROWELL

ABSTRACT. Software is a fundamental and integral part of many medical devices. While the doctor who performs an operation must be licensed, the developers who create the software used in these devices undergo no such licensing. For this reason it is especially important that designs which can be subjected to rigorous analysis be produced prior to software development. Such designs are fundamental to effective and efficient testing, analysis of systems-of-systems, and risk determination and mitigation. This paper identifies challenges and potential research directions to address these concerns.

1. MOTIVATION

In [3] the authors discuss the well-known Therac-25 accidents which resulted in death and serious injury, and identify basic software engineering principles that the Therac software designers violated:

- Documentation should not be an afterthought.
- Software quality assurance standards should be established.
- Designs should be kept simple.
- Ways to get information about errors—for example, software audit trails—should be designed into the software from the beginning.
- The software should be subjected to extensive testing and formal analysis at the module and software level; system testing alone is not adequate.

Three of these principles mention documentation and design, and one mentions formal analysis of the design. Unfortunately these principles continue to be violated for a variety of reasons:

- Software requirements are poorly-understood at the outset and change as developers and customers learn about the system's capabilities. Thus early rigorous design work is often considered wasteful, since the design will inevitably change.

S. Prowell is with the Department of Computer Science, The University of Tennessee, 203 Claxton Complex, 1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA. Email: sprowell@cs.utk.edu.

- Many software developers are unfamiliar with modern formal specification techniques and tools.
- As Parnas and others have noted [4], many of the existing formal approaches focus on notation, but not content.

The notion that early design work is wasteful ignores the established idea of incremental and iterative development [2] under statistical process control. This idea can be applied to the specifications as well as to the software, itself. Thus the first objection above is more a matter of educating developers than of developing new techniques.

Software is a fundamental and integral part of many medical devices, even where the software aspect of the device is not obvious (as with embedded systems). Safety and regulatory requirements for software systems must require rigorous verification and validation to ensure adequate reliability to protect the public health. As software systems become increasingly complex and interconnected, testing as the means to gain confidence in the system necessarily becomes impractical. Further, the lack of a precise system specification inhibits even the best testing approaches, and makes it difficult for domain experts (cardiologists, neurosurgeons, etc.) to ascertain whether the device will perform appropriately even if it has been thoroughly tested.

While the doctor who performs an operation must be licensed, the developers who create the software used in these devices undergo no such licensing. For this reason it is especially important that designs which can be subjected to rigorous analysis be produced prior to software development. Such designs are fundamental to effective and efficient testing, analysis of systems-of-systems, and risk determination and mitigation. This paper identifies challenges and potential research directions to address these concerns.

2. CHALLENGES

Three challenges inhibit the adoption of rigorous specification techniques:

(1) New methods. We require straightforward, systematic methods to create rigorous specifications from informal, poorly-understood, and often incomplete and inconsistent initial requirements. These methods should be *notation independent* where possible, to allow the maximum use of existing tools and techniques.
(2) Understanding of the state-of-the-art. Where tools and notations exist, their role in the engineering process, their limitations and applicability, and their fidelity must be understood. Especially when

software is combined into systems-of-systems, the impact on underlying assumptions of the methods must be understood. As significantly, the cost in terms of time and money for these methods must be understood in order to adequately plan for and justify their use.

(3) Barriers to adoption. Existing practitioners must be educated on the methods and tools, and curricula must be developed for software engineering programs. If existing practitioners and students do not understand the relevance and utility of the methods, they will not be adopted.

Each of these challenges is of a different nature, and these challenges are neither finite nor sequential, but define an ongoing research program.

Tools such as model checkers and theorem provers, coupled with formal notations such as Z [6] and CSP [1], provide a way for developers to create and analyze software designs prior to implementation. Unfortunately, if the methods used to create designs in these notations are poorly-understood, then writing a design in Z or CSP becomes just another kind of programming. Approaches such as [5] can be applied to generate a traceable specification from initial requirements, and the result can be transformed into a variety of different notations for analysis using existing tools.

The most significant of all these challenges is the education challenge. If software developers do not see the relevance and utility of the techniques, they will resist adoption, and will "work around" these techniques. This leads back to the first of Leveson and Turner's basic principles: documentation should not be an afterthought. Unless documentation contributes meaningfully to the process and is seen as helpful, it will continue to be an afterthought. Thus Parnas' argument that the content of documents matters *more* than the notation.

3. Moving Forward

There seems to be a common misconception that formal methods cannot be widely adopted and used for developing software. This is demonstrably false. Rate-monotonic and deadline-monotonic analysis techniques are formal, and are widely-adopted to create real-time schedules. The construction of parsers is based on formal grammatical notions. These formalisms are then hidden by well-understood methods and tools, which free developers to work at a higher level of abstraction. The tools and notations are understood to contribute, and are thus widely adopted. Another example is database normalization. The importance of normalization is understood by developers and thus the normalization rules are widely adopted.straightforward

Methods which result from the research program must meet this "adoptable" criterion and developers must be actively educated in their use and

applicability. Not every technique is universally applicable. Engineers must understand not just how a technique helps, but under what conditions it will help. Managers must understand the cost and benefits of each technique. Such an analysis is difficult to do. In order to facilitate this, a protocol for sharing the results of applying the techniques must be established. Because companies have little incentive to share good techniques with their competition, there is an important role for regulatory agencies in collecting and evaluating the results of applying these techniques.

Moving forward, the following approach is suggested:

(1) Survey fundamental theory relevant to the specification of software systems. It is likely that current theory is sufficient to the specification of modern software systems.
(2) Identify practical, straightforward, and teachable practices based on sound theory. The theoretical assumptions determine the applicability of the practices, and guarantee the correctness of their results.
(3) Implement tools which support the application of these practices and connect them directly to the generation of product code.

REFERENCES

[1] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.
[2] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, June 2003.
[3] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.
[4] David Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25:41–61, 1995.
[5] Stacy J. Prowell and Jesse H. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29(5):417–429, May 2003.
[6] Mike Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, New York, 2nd edition, 1992.