LESSONS FROM ENGINEERING
Can Software Benefit from Product based Evidence of Reliability?

Neal Snooke
Department of Computer Science, Aberystwyth University, Penglais Campus, Aberystwyth, U.K.

Keywords: Software engineering, Model-driven engineering, Reliability, Failure modes and effects analysis, Verification.

Abstract: This paper argues that software engineering should not overlook the lessons learned by other engineering disciplines with longer established histories. As software engineering evolves it should focus not only on application functionality but also on mature engineering concepts such as reliability, dependability, safety, failure mode analysis, and maintenance. Software is rapidly approaching the level of maturity that other disciplines have already encountered where it is not merely enough to be able to make it work (sometimes), but we must be able to objectively assess quality, determine how and when it can fail and mitigate risk as necessary. The tools to support these tasks are in general not integrated into the design and implementation stages as they are for other engineering disciplines although recent techniques in software development have the potential to allow new types of analysis to be developed and integrated so that software justify its claim to be engineered. Currently software development relies primarily on development processes and testing to achieve these aims; but neither of these provide the hard design and product analysis that engineers find essential in other disciplines. This paper considers how software can learn from other engineering analyses and investigates failure modes and effects analysis as an example.

1 INTRODUCTION

Software has become extremely complex and pervasive in a very few years, and the engineering techniques to support its design and implementation are lagging behind, resulting in many well documented project failures, fragile and brittle systems, user frustrations, workarounds and occasionally hazardous systems. Software is taking over many roles that only a decade or so ago were performed by mechanical electrical, hydraulic devices. In commerce and everyday life we now depend on software performing a diverse range of tasks from maintaining critical medical and financial information to entertainment systems. This paper proposes that engineering techniques used to develop software have not kept pace with the role that software now plays, and in particular most software does not provide satisfactory *product based evidence* of its quality and safety.

In other disciplines, engineers routinely address many non functional concerns such safety, quality, reliability, robustness, maintainability, diagnosability, degradation and prognosis to name a few. These concerns are directly addressed at the design and implementation level and analysis techniques are used that integrate into the design process and subsequently to allow direct assessment of the product performance. It is not merely enough to be able to make it work; to attain the status of a properly engineered product we must be able to objectively assess its quality, determine how and when it can fail and mitigate as necessary. These capabilities are necessary when any complex product is to be built to a specified quality, within time and cost constraints.

Although the nature of software is different to other engineering disciplines, there are many similarities and analogies that could be exploited to improve the software engineering endeavour by developing analytical reliability engineering techniques. As new software development abstractions and technologies are developed the opportunity to support additional engineering analysis should not be missed in the rush to provide ever greater functionality at the expense of poorly engineered products of extremely variable and unknown quality.

The remainder of the paper will discuss the weakness of current software development compared to other engineering development. Several kinds of reliability analysis that might be adapted to software are then considered together with the developments

Snooke N. (2010).
LESSONS FROM ENGINEERING - Can Software Benefit from Product based Evidence of Reliability?.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 238-244
DOI: 10.5220/0003039102380244

in software technologies that are (or will) make such analysis feasible. Finally we will consider software failure modes and effects analysis in more detail as an example of an analysis that might be adapted to software, even though on the surface it appears that there is no need since software does not age or wear out in the same way that hardware components do.

2 THE LIMITATION OF DEVELOPMENT PROCESS AND TEST

The achievement of software quality levels is almost exclusively based around standards that define development *processes* and practices. While development processes are vitally important (in any engineering endeavour), as observed by McDermid (McDermid, 2001) they do not guarantee the safety of a product. If fact even non safety critical software would benefit from product based *evidence* of reliability and quality characteristics.

Testing is the primary (and often the only) evidence based analysis performed on software, and although testing can *initially* give indications of quality, once the results are used to improve the software, the tests are no longer a reliable indication of overall quality. The value of tests in assessing quality is further reduced by hierarchical testing because lower-level modules will themselves have been made to pass a set of tests before being included in a system. It is likely that the system outside the scope of the tests performed is near the quality achieved at the very first execution of the code (McDermid, 2001). Put another way, testing only finds faults, it does not help prevent or assess them. The reason for this is clear. Tests can only ever verify a tiny sample of the expected behavioural envelope of the software and they are often focused on verification of nominal function, with perhaps a few limiting case examples. Other domains such as electrical systems analysis also have an infinite number of possible numerical behaviours, but for example, failure effect analysis is carried out by abstracting these behaviours into qualitatively similar regions, and concentrating on a worst case analysis. All substantial software will enter unanticipated regions of behaviour during its life and all software will have to deal with external failures either from hardware or other software, and for most software there is limited understanding of how the system might behave.

Traditional formal methods are often proposed as the solution to these problems, but they can be difficult to use, specification capture remains an issue, and pragmatically these methods are too expensive for the majority of software. Software Integrity Levels are often used to demonstrate a specific level of integrity(IEC61508, 1998) however SIL level specification most often results in a mechanism to change product requirements into process requirements and for higher levels may also require formal methods. These mechanisms are typically very expensive to implement and accordingly are used only for the highest risk systems, leaving the vast majority of software development relying solely on the talent and intuition of software developers with a little testing to verify basic functionality.

Testing covers a wide range of analysis, some of which begin to move towards a wider ranging type of analysis that can consider both more comprehensive and/or abstract behavioural considerations, for example functional analysis testing and cause-effect graph testing. Other types of testing such as statement coverage, path coverage, and modified condition decision coverage can help discover obvious inadequacies in test suites but they cannot reason about overall behaviour and potential failure modes or problematic operating states. As far back as 1992 (Voas, 1992) proposed an analysis that identifies where faults *if they exist* are more likely to remain undetected, and although based on execution of test sets it in the spirit of a broader assessment of quality.

Testing provides a very precise and accurate description of behaviour but as a result has low coverage. What is missing are the lower precision but higher coverage techniques, that can reveal more general (good and bad) characteristics and *potential* problems.

Notice that lower precision does not imply inaccurate. For example a simple analysis could abstract the value of an input to good or faulty and propagate the result to affected outputs of the system {good, definitely faulty, possibly faulty}. The analysis will be accurate if it correctly categorises all outputs. At this level we may not care about internal state, if it can be shown that the fault can reach an output then the system function(s) associated with the output are suspect for this fault. Such an analysis could allow undesirable characteristics of a system to be determined, for example if the effects of hypothetical faults are presented in terms of the system functions, any faults that do not propagate within the input/output relationships expected from system functions may be a concern, particularly if the function is core or critical to the application.

Notice the need in this example for some form of functional description of the system that is formalised

enough to be used by an analysis tool. This is essentially capturing higher level design knowledge to assist the analysis, and techniques such as the Unified Modelling Language (UML2) and Model Driven Architecture (MDA) are starting to provide the higher level information that will facilitate such a range of analysis methods. As these modelling techniques are developed the potential for design analysis should be considered alongside the main focus of system specification and (automated) code generation.

3 KINDS OF ANALYSIS

Reliability engineering is concerned with ensuring a system can perform its required functions for the required period of time. Reliability should be considered as part of both verification and validation activities since it is necessary to verify both that the design supports reliable provision of the functions of the system via inherent (or architectural) reliability, and subsequently to validate that the implementation provides satisfactory failure characteristics consistent with the expectations of the high level design. This section will consider one area of validation that often lacks a comprehensive analysis in software systems - the failure characteristics of the system.

Software codes do not wear out or suffer manufacturing faults and since in essence software is pure logic, it is easy to be seduced into believing that with testing it will be perfect. In contrast, all physical system engineers accept that both their product and the environment it operates in will be subject to failure, and experience shows that - although it is less readily accepted - this is also the case for software. Faults in physical systems are either due to manufacturing or wear out, however faults in software are due to built in flaws or unanticipated (hardware or software) operating environments. The changing environment of software often gives the end user the appearance of ageing faults where parts of the software no longer work properly and we are all familiar with the endless updates that often then introduce new problems. Of course the software never worked in the environment that caused the failure, but to the user the environment may not appear any different, and even if it is, the phrase “I don’t see why it should affect that...” is commonplace from both end users and technical support staff. Probably the software was inherently fragile in either its design or implementation and because the testing only represented one specific instance of a possible environment, a minor problem can propagate and cause apparently unrelated failures.

For hardware devices it usually is possible to provide tests that will detect the vast majority of manufacturing defects, however for software it is impossible to provide sufficient tests to have anywhere near the same level of confidence. The operating environment of software is also far more complex and difficult to specify compared to mechanical or electrical systems (for example). The lack of wear-out makes maintenance of software a much worse problem since code can exist for an indefinite period of time and continuously form parts of new products. Of course it is really the design being reused but for the reasons above it is easy to reuse software in an environment that uncovers latent faults.

The presence of wear-out faults ensures that hardware engineers must use designs that are inherently robust against failure and that failures are contained as far as possible, and that in any case the worst effects of failures are assessed at design time. Unfortunately the lack of wear-out removes the focus from these tasks in software design/implementation, although clearly better consideration would lead to higher quality software. Analysis methods that assess failure mode behaviour include:

- Failure Mode and Effects Analysis (FMEA). FMEA is an inductive bottom up approach that analyses component (or input) faults on the behaviour of the system and its functions. SFMEA is used to mean and FMEA targeted at software to distinguish it from software that performs or assists hardware FMEA. It is a comprehensive analysis that considers large numbers of low level faults and categorises the effects according to risk based on the severity of functions potentially affected and the detectability of the fault. The engineer will assess the highest risk faults, and faults that have unexpected consequences.
- Fault Tree Analysis (FTA) is a top down approach that deduces the possible causes of undesirable top level events. Fault tree analysis is good at dealing with multiple faults, but when done manually it is not good at finding all possible faults.
- Reliability Block Diagram is a diagrammatic method for showing how component reliability contributes to the success or failure of a complex system. RBD is also known as Dependence Diagram or DD.

The ability to perform such sophisticated analyses relies on the fact that most faults lead to a specific behaviour or class of behaviours. Software, however, can have the highly undesirable characteristic that any fault can lead to almost any effect, however higher level programming languages and modelling allow the compilers and transformation tools to pro-

vide behavioural and structural constraints that combined with good system design limit failure modes resulting in the possibility of a meaningful analysis. Of course the tools used to transform models into software must correctly enforce the required computational infrastructure, but as for current compilers, widespread use, the relatively small and cohesive operations involved and the mathematical formalisation used ensures high reliability.

4 FMEA AS AN EXAMPLE

This section will outline the emerging software characteristics that will facilitate the ability to objectively analyse software reliability at design and implementation time using an FMEA approach.

A number investigations into Software FMEA having been carried out, for example (Ozarin and Siracusa, 2002; Nguyen, 2001; Bowles, 2001), however such endeavours have been largely a manual task, extremely tedious and expensive in terms of engineer time, and only justified for the most safety critical applications. Typically these analyses have been a response to the requirement for a system FMEA in situations where the system contains software. The concept of an FMEA focussed directly at software would be valuable but will only be feasible when the analysis can be carried out automatically and the software engineer presented with a meaningful set of concerns.

Interest in SFMEA has been steadily growing and some formal guidelines have been developed for example the SAE G-11 RMSL division Software Reliability Subcommittee extending the Software Reliability Program Implementation Guide SAE JA1003 (SAE-JA1003, 2004). However it is not the task of those guidelines to develop new software analysis techniques, and and the focus is on ensuring the best available tools and practices are used to provide the required level of quality.

In general the aim is *not* to produce an error free system, but to demonstrate that the fault behaviour pertaining to both external and internal potential faults is understood and constrained. Therefore if an software bug causes a module to produce an incorrect result, the FMEA will have determined how severe the impact on the system could be, and have allowed actions to be taken to mitigate high risks. For safety related systems the aim is only to ensure safety, for other systems the aim is to ensure uniform quality and reliability, so that for example faults in non essential functionality do not cause serious failures in core functionality. Using a concrete scenario, faults that can lead to loss of user entered data will typically be higher risk than faults that simply require software to be re executed, however risk is very much dependent upon the specific application level functionality. Potentially critical faults, critical modules or unexpected failure modes can be identified and validation activities focussed. Implementation mistakes that lead to undesirable or unexpected failure modes can be identified.

An FMEA is designed to extract specific abstract characteristics of a system whilst basing its findings on the details of the system design and implementation. It therefore has several properties that lead to a different kind of analysis to those generally applied to software:

- an FMEA aims to locate the *worst case* effects of a comprehensive set of hypothetical faults or system failure modes.
- FMEA does not verify functionality - that *is* a primary role of testing - and an adequately functioning system or design is the starting point.
- the aim is to identify failure modes. These are characteristic ways the system can fail, usually based in the way in which systems functions are affected.
- Regardless of how much range and consistency checking is done it will always be possible for an incorrect but valid input to exist, and it is important to know how badly it may affect the system, such as which outputs can be affected and if any longer term internal state might be corrupted.

These features imply that an abstract analysis is required. For example data values may be considered not as specific values but just as $\{\mathsf{faulty}, \mathsf{nominal}, \mathsf{possibly\_faulty}\}$, if a behaviour data flow analysis is being carried out using the actual code then all affected data paths and affected control paths should be considered. For numerical variables a qualitative representation may be suitable, this approach has worked well for automated FMEA generation of complex electrical systems for example (Price et al., 1997; Lee and Ormsby, 1993) that allow prediction of worst case effects for whole regions of system behaviour since even non-software systems may have too many parameters and attributes to test all numerical value permutations. Software presents an even greater challenge in this respect since it contains vastly more non-linear behaviours and states however techniques such as abstract interpretation and model checking provide formalisms, particularly if software is developed via modelling techniques that explicitly support the techniques by limiting unnecessary complexity. The key will be to design and model in such a way that complexity is constrained, hierarchical and compositional.

A way of interpreting the detailed analysis results is required, so that failure modes can be identified. The functions of a system are ideal since they have a close link to the original requirements. The functions must be captured in a form where they can be used by an analysis tool. One such approach is the Functional Interpretation Language (FIL)(Bell et al., 2007) that has been successfully used to build hierarchical functional models for electrical systems, and should be general enough to apply to software. Such methods are designed to provide a formalised framework that allows automated reasoning, while also providing an intuitive way for engineers to provide application information. Some of the information is already available, for example UML use cases identify user level input/output relationships that have a close resemblance to the trigger/effect relationships that identify function state (inoperative, achieved, failed, unexpected) in the FIL.

Code execution is neither feasible nor required, and specific values are not considered unless they have a specific and distinct behavioural significance. The analysis can be carried out at a number of levels from system block diagrams, to state charts to the code itself providing levels of effect detail. High level analysis can demonstrate the effects of architectural choices on potential failure behaviour, and low levels can both verify that implementation details have not modified the high level effects, as well as identifying specific local vulnerabilities. Lower level or code analysis must make use of pseudo-static methods that abstract behaviour as necessary for failure analysis. In an investigation into automated SFMEA (Snooke and Price, 2008), the code is not executed and modified functional dependency modelling (Chen et al., 2004; Chen and Wotawa, 2003; Mateis et al., 1999) and data path analysis combined with shape analysis (Corbett, 2000; Sagiv et al., 1998) that abstracts out the form of data structures allow worst case effects to be considered. To take an example, if some code implements a list, an error in the data value in the list will be predicted to propagate into any code that accesses the list. It is not necessary to consider where (or even if) in an execution cycle it will happen since sooner or later it can. If a failure mode cannot happen for some (application specific, or logical, or value related) reason then documenting the implicit constraints and adding the relevant checks into the code (possibly automatically) will ensure that the system continues to exhibit the required and tested behaviour.

We therefore envisage an analysis that does not actually execute the code but is able to reason about worst case analysis in the presence of faults. Clearly the technicalities of such an analysis are complex and some types of coding are more easily amenable; for example embedded code that does not allow dynamic data structures is clearly much easier to analyse. However as modelling and languages develop more sophisticated ways to design software and prevent programmer errors the constraints imposed also make the analysis easier.

The failure modes of software based systems have been considered by several studies (Raheja, 2005; Czerney et al., 2005; Goddard, 2000). From these there are a set of generic failure modes associated with software, many of which are catastrophic at the level of a software process such as failure to execute, executes incompletely, failure to return from ISR. Others such as coding errors, logic errors, I/O errors, external hardware/software failure, definition of variables, omissions in the specification, insufficient memory, operational environment, resource conflict are general and require specific instances to be considered within the context of system functions and implementation. Hardware failures such as EMI/RFI, power outage, corrupted memory, loose wires and cables are not the concern of the software design, but rather of the system architecture and can be addressed via existing hardware design techniques.

Other systems have similar distinctions for example failures in electrical systems may cause catastrophic short-circuits and unexpected thermal overload in the wires of the system, but most affect only (loss of) functionality. Design rule checking and good design practice helps alleviate many of the catastrophic problems in electrical systems and in software we have exception handlers, strong typing and other widely used techniques to prevent catastrophic execution failures. Functional failures are currently much more difficult to assess and detect in software, and such an analysis only makes sense once the system structure and behaviour is somewhat localised. The application specific failure modes then become specific combinations of function failure, possibly including an abstraction of the mechanism that causes the failure.

In electrical systems physics provides fault localisation and results in limited effect propagation and therefore most faults do not lead to total failure. In software at the lowest level almost any fault is like this and can also lead to almost any effect. For example writing in assembly language on a Von Neuman architecture computer a fault leading to data being written to the wrong address can lead to any effect when that data is interpreted as an instruction.

Physical constraints such as spatial partitioning, mass, manufacture cost and material strength considerations tend to limit structural complexity, leading

to layers of structuring and separation of functions which in turn limits fault impacts. One of the difficulties with complex software systems is the relationship between faults and effects. A minor fault can, for example, cause cascading errors in a software system or have almost invisible but very complex, subtle, and long lasting side effects. The result is that software often has very non-uniform quality in terms of the effects of potential failures, and it is not clear where effort should be expended to improve quality.

Modern software languages both encourage and enforce higher levels of structuring and enforced properly by compilers this helps to inherently constrain faults. Decades ago, high level-languages introduced typed data and procedures to help structure data and code and these have been refined ever since. OO methods provide common structures to partition data with code. Aspect oriented programming further abstracts cross cutting concerns such as logging, and security and perhaps allows separate analysis of these concerns, similar to the electrical, mechanical and hydraulic domain subsystems of products are analysed. Gradually a computational infrastructure is emerging that (once it is trusted) provides the analogy to the physics of the engineering world. These techniques make SFMEA analysis feasible in that they constrain potential fault impacts.

As an example, structural constraints imposed by the language can have a strong effect on the propagation of detailed faults. For example, in figure 1, a language has been used that separates memory into data and instructions and prevents data from being executed. Moreover, structuring of the instructions ensures that faults in some instructions can only affect some output (functionality). The language also allows the data memory to be partitioned by typing and data structure separation, preventing faults in some locations from propagating to others. Conversely languages that support facilities such as arbitrary pointer arithmetic allow faults to propagate very widely, seriously reducing FMEA efficacy.

However analysing even OO language code for failure propagation is a difficult task, and the hope is that approaches such as MDA may improve the situation and future developments will directly support failure and other types of engineering analysis.

5 CONCLUSIONS

This paper argues that reliability techniques should be developed that are of use in general software as well as for traditional safety critical applications. Some of the motivation does come from application areas

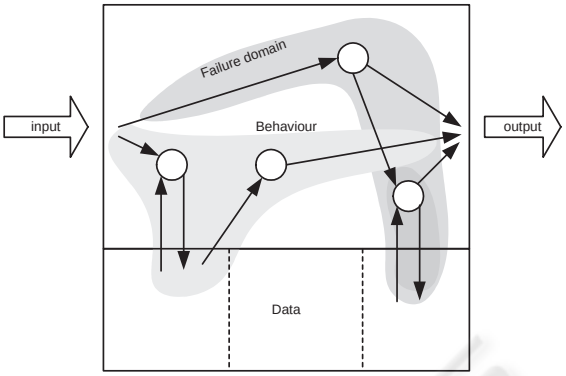

Figure 1: Failure domain localisation in software.

such as the automotive environment where software is replacing hardware and there is a need for reliability analysis for software equivalent to that once performed on the hardware. The engineers working in these domains are used to using many types of analysis to improve non functional product aspects. The second part of the motivation comes from the observation that software is now very complex, with many layers of abstraction that both provide the potential to allow improved analysis but also provide the potential for serious and difficult to find latent failure modes.

The fundamental assumption is that complex software will encounter faults of one form or another, from coding errors, to specification oversights and assumptions about the operating environment; and we need to be able to quantify the potential impacts of faults and ensure acceptable behaviour in response to faults. It is only with the advent of higher level programming, specification and modelling techniques that fault behaviour is constrained enough, to make such an analysis feasible and hopefully the situation will improve, particularly if new languages and modelling provide a computational infrastructure that has analysis designed-in rather than added-on. As pointed out by (Leveson, 2004) characteristics such as safety and reliability are emergent properties, and analysis of such properties is possible for physical systems because they can be analytically reduced. Feedback loops and nonlinear behaviour create hard to analyse emergent behaviour, therefore metamodelling that limits the use of these characteristics and makes explicit the intended effects would be essential to facilitate non functional design analysis tasks. Some work is already being done in the area of improving the utility of higher level modelling, for example (Iwu et al., 2007) proposes a Practical Formal Specification that integrates with UML in the area of safety critical development, however there is a need for improved modelling with semantics that allow a range of low effort light weight automated analysis for everyday

software.

This paper only sketches a vision, however there are already many techniques available that can be connected to develop novel types of analysis for some languages and modelling environments. Engineering reliable software needs to be a combination of a culture where failure modes are accepted, foreseen and assessed combined with design and implementation formalisms that facilitate automated analysis and verification. This does not mean formal methods in the traditional sense but rather flexible and expressive modelling, metamodelling, and specification techniques that are accessible to application engineers but also have with semantics and syntax that can be processed by sophisticated tools. The tools that perform the analysis must present the engineer with concise and easily understood explanations of issues and for this to happen software engineering must consider a wider range of analysis options than syntax checking and testing.

REFERENCES

Bell, J., Snooke, N. A., and Price, C. J. (2007). A language for functional interpretation of model based simulation. *Advanced Engineering Informatics*, 21(4):398–409.

Bowles, J. B. (2001). Failure modes and effects analysis for a small embedded control system. In *Annual Reliability and Maintainability Symposium*, pages 1–6. IEEE.

Chen, R., Kob, D., and Wotawa, F. (2004). Exploiting static abstraction of data structures for debugging. In *MONET Workshop on Model-Based Systems at ECAI 2004*, Valencia, Spain.

Chen, R. and Wotawa, F. (2003). Debugging with an enriched dependency-based model or how to distinguish between aliasing and value assignment. In *Proceedings of the International Workshop on Qualitative Reasoning (QR-2003)*, Brasilia, Brazil.

Corbett, J. C. (2000). Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93.

Czerney, B., D'Ambrosio, J., Murray, B. T., and P.Sundaram (2005). Effective application of software safety techniques for automotive embedded control systems. In *SAE 2005 World Congress and Exhibition*.

Goddard, P. L. (2000). Software FMEA techniques. In *Reliability and Maintainability Symposium*, pages 118–123. IEEE, IEEE.

IEC61508 (1998). *Functional Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems (IEC 61508)*. International Electrotechnical Commission, International Electrotechnical Commission, 3 rue de Varembé, Geneva, Switzerland, http://www.iec.org.ch edition.

Iwu, F., Galloway, A., McDermid, J., and Toyn, I. (2007). Integrating safety and formal analyses using UML and PFS. *Reliability Engineering and System Safety*, 92(2):156–170.

Lee, M. and Ormsby, A. (1993). Qualitative modelling of the effects of electrical circuit faults. *Artificial Intelligence in Engineering*, 8:293–300.

Leveson, N. G. (2004). A systems-theoretic approach to safety in software-intensive systems. *IEEE Trans. on Dependable and Secure Computing*, 1(1):66–86.

Mateis, C., Stumptner, M., and Wotawa, F. (1999). Debugging of java programs using a model-based approach. In *10th International Workshop on the Principles of Diagnosis (DX'99)*, pages 166–173.

McDermid, J. (2001). Software safety: Where's the evidence? In *6th Australian Workshop on Industrial Experience with Safety Critical Systems (SCS '01)*. Australian Computer Society. Available: http://www-users.cs.york.ac.uk/ jam/.

Nguyen, D. (2001). Failure modes and effects analysis for software reliability. In *Annual Reliability and Maintainability Symposium*, pages 219–222. IEEE.

Ozarin, N. and Siracusa, M. (2002). A process for failure modes and effects analysis of computer software. In *Annual Reliability and Maintainability Symposium*. IEEE.

Price, C. J., Pugh, D. R., Snooke, N. A., Hunt, J. E., and Wilson, M. S. (1997). Combining functional and structural reasoning for safety analysis of electrical designs. *Knowledge Engineering Review*, 12(3):271–287.

Raheja, D. (2005). Software FMEA: A missing link in design for robustness. In *SAE 2005 World Congress and Exhibition*. SAE International.

SAE-JA1003 (2004). *Software Reliability Program Implementation Guide*. Society of Automotive Engineers, http://www.sae.org/technical/standards/ja1003_200401 edition.

Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50.

Snooke, N. and Price, C. J. (April, 2008). An automated software FMEA. In *Proc. International System Safety Regional Conference (ISSRC 2008)*, Singapore.

Voas, J. M. (1992). Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18:717–727.