Supporting Reliable Software Evolution through Program Analysis

Cathal Boogerd
Software Evolution Research Lab
Delft University of Technology
The Netherlands
c.j.boogerd@tudelft.nl

Abstract

This paper motivates the need for more research to ensure a consistent level of reliability in software systems. We briefly outline the relevant developments that drive this need, define the necessary concepts, and sketch the context of the proposed research. We then define the research space by introducing three research questions and mention some first ideas of solutions to the problems they represent. Finally, we discuss the progress made so far in more detail, and conclude with a description of validation approaches and expected contributions.

1. Introduction

Software evolution has long been known to be a major expenditure in the software life cycle [10]. Yet recent developments, notably in the area of embedded systems, stress its importance once more. We have witnessed a tremendous growth in the amount of software embedded in different high-tech products: various features, previously implemented in hardware, are now implemented in software for eased extendability and adaptability. In other words, the ability of software to evolve provides new business opportunities.

However, another result is that the reliability of the embedded software has become a prime concern for their manufacturers. In addition to software complexity, there are two other trends contributing to this concern. First of all, the market for these products is highly competitive, so time to market is rapidly decreasing, further pressuring development. Secondly, these systems typically need to interface with many third party systems, leading to unforeseen issues with regard to security and reliability.

Two key issues emerge from this sketch: a volatile environment requires software that can be quickly adapted to changing circumstances, or be *evolvable*. However, this flexibility must not compromise system *reliability*.

Reliability and Evolution IEEE [1] defines software reliability as "The ability of a system or component to perform its required functions under stated conditions for a specified period of time.". Although there does not seem to be a common definition for software evolution, here we will use it as a preferred alternative to *software maintenance*, defined in the IEEE Glossary as "The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment".

The two terms meet in the concept of *software change*, as reliability is one of the concerns that may prompt evolution of a system. Conversely, making changes to existing software can have a 'ripple effect' in that system [16], introduce new defects, and thereby impact reliability. We observe that efficient management of a system's evolution is instrumental in ensuring continuing reliable operation. As such, the research proposed in this paper seeks to develop methods, tools and insights assisting development teams in managing their system's evolution, with an emphasis on reliability.

Context This research is carried out within the Trader project [8], in which several academic and industrial partners collaborate in creating design methodologies, development tools and run-time safeguards for maximizing system reliability. Furthermore, Trader specifically recognizes the importance of *user perceived* reliability, with one track being devoted entirely to developing a model capturing user behavior in this regard. Currently ongoing research deals with topics such as reliability assessments through error injection, error detection, fault diagnosis, and software architecture assessments. These tracks are complementary to our own, which will be discussed in greater detail in the next section.

2. Goals and Approach

As software change is the concept impacting both evolution and reliability, a good starting point for discussion of the different tracks identified in this section is the software change mini-cycle [16], consisting of the following phases:

- *Request* This can be a request for implementation of a new feature, or a bug report.
- *Planning* This phase entails program comprehension to the extent necessary for the change, and understanding its impact.
- *Implemention* Here the developer restructures the code, and propagates any changes necessary throughout the code.

- *Validation* Ensuring that the bug is fixed or the new feature works correctly, and no new defects are introduced. This may include regression testing or automatic code inspection.
- *Re-documentation* The understanding gained during the comprehension phase can be consolidated in updated documentation.

Research Questions Our research will be focused on aiding the developer in the planning, implementation and validation phases of a change, thereby supporting them in the evolution of their software. At present, we distinguish a number of different directions, targeted at reliability, represented by the following research questions:

RQ1. *How to identify reliability issues in software?* Providing an answer to this question can help developers in change impact analysis, change propagation, and finally, the validation phase. In order to answer this question, we need to find an answer to the related question: what are reliability issues? An approximation of the answer can be found by observing patterns in the code known to be potentially related to undesired behavior. We can distinguish between the more generic issues, such as the ones typically mentioned by software inspection tools (cf. section 3), and more domain-specific ones, that can be found by in-house solutions or customizable tools. We believe that an extensible classification of reliability issues and their corresponding source code patterns will be of great help in understanding commonalities, forming the basis for new and improved detection mechanisms. Such patterns are similar to the well-known *code smells* [5], yet we would like to define them at a higher level of abstraction than the typical code smell. The classification must allow for extension in order to benefit from domain-specific knowledge, that may bring in its own known issues and patterns. Such a classification may be obtained from a combination of literature study, analysis of existing tools, and an investigation of the fault tracking databases of our industrial partners.

RQ2. *How to establish an ordering of reliability issues in terms of impact and severity?* This is an increasingly relevant topic because of one of the reasons mentioned in the introduction: development time is decreasing, so situations might arise in which we do not have enough time to solve all known issues. This is especially true, as the tools employed in finding such issues typically produce a conservative approximation of the actual set of issues present, i.e., list *false positives* amongst their reports. Combined with a significant codebase, this leads to an explosion in the number of issues to be dealt with, forcing developers to resort to all kinds of (manual) filter processes. Even worse, discussions with our industrial partners indicate that the information overload often results in complete rejection of the tool. Therefore, we need to have an idea of the relevance of every issue to determine which one should be addressed first. This has been the focus of our earlier work, which is discussed in more detail in section 3.

RQ3. *How to improve software fault tolerance?* This is a very broad question, but we limit the scope by focusing on the handling of exceptional situations in source code. In languages that do not provide explicit constructs for handling exceptions, such as C, these are usually implemented in an idiomatic way, e.g., using return codes. We want to reconstruct an explicit exception handling model from these implicit mechanisms in the source code, identifying weaknesses in exception propagation flows, and finding locations where a more explicit way of handling will allow more accurate pinpointing of the actual cause of an exceptional situation. When we have obtained the necessary exception handling models, we can use those to migrate the existing implicit or incoherent handling mechanisms to a unified, explicit structure. For instance, we can create an exception handling aspect, and use code transformation techniques to (semi-)automatically adapt the source code to an AOP solution.

Approach The principal technique employed in answering our research questions will be *program analysis*. Program analysis denotes a collection of techniques that extract information implicitly present in a program. Usually, these are subdivided into *static* analysis, analyzing only the source code of the program, and *dynamic* analysis, incorporating information retrieved from program runs. Presently, we focus on static analysis, but eventually, we want to try hybrid approaches. We propose to use source code analysis for a variety of reasons:

- *Availability.* In constrast with many of the other software process artefacts, such as higher-level documentation, design decisions or architecture specifications, the source code is readily available and up-to-date. This is especially true in case of legacy systems, where often both the original development team and the appropriate documentation are lacking.
- *Flexible Use.* Techniques based on program analysis can be used at any stage in the development process, as long as there is source code to work with. This allows for frequent use and therefore continuous monitoring of software reliability.
- *Adoption.* Solutions based on program analyses can be employed (semi-)automatically, and do not necesarily require introduction of extra artefacts, such as large sets of test data, descriptions of operational profiles, etc. Consequently, this allows us to come up with techniques that require little developer intervention, thus imposing only a small overhead on the development process, which should ease the actual adoption.
- *Code Ergo Est.* This playful variant on Descartes' famous saying hints at the conjecture that software properties, including reliability, are ultimately determined by its source code. It makes sense, therefore, to use source code as a primary source of information, supplemented

by any other artefacts available, and this is exactly what program analysis does.

The next section will discuss the progress made so far in answering our research questions.

3. Preliminary Work

The work discussed in this section has been previously published in [3], where we assume that reliability issues are found in the form of defects in the source code, by means of software inspection. Experiences reported by our industrial partners and in literature [9] indicated that the high number of false positives generated by existing inspection tools greatly limit their usability. It is for that reason we first started addressing research question RQ2.

Software inspection [4] is widely recognized as an effective technique to assess and improve software quality and reduce the number of defects [12, 6, 14, 11, 15]. Software inspection involves carefully examining the code, design, and documentation of software and checking them for aspects that are known to be potentially problematic based on past experience.

Our approach deals with tools that perform automatic code inspection. Such tools allow early (and repeated) detection of defects and anomalies which helps to ensure software quality, security and reliability. Usually these are built upon some form of static analysis, ranging from a compiler in pedantic mode to fully-fledged, customizable tools such as QA-C[1] or Codesonar[2]. For reasons mentioned in the previous section (under RQ2), these tools usually report a large number of issues. This results in developers having to manually filter large lists of reports, and often completely rejecting the tool.

Recall the context of our work discussed in Section 1, and the notion of *user perceived* reliability. Bearing this in mind, we would like to concentrate on the issues that produce the most noticeable behavior. A fundamental requirement is that the code associated with a given issue must be executed in order to produce any visible behavior at all. Therefore, we propose to use the *execution likelihood* as a notion of relevance, ranking the reported issues with respect to the probability that control flow will reach the associated program statements. The result is a *generic prioritization approach* that can be applied to the results of any software inspection tool and assists the developer in selecting the most relevant warnings.

Voas et al. [13] define a related notion, *software testability* as "the probability that a piece of software will fail on its next execution during testing if the software includes a fault." They present a technique that estimates three metrics for every location in the program: the first one is *execution probability*, similar to our execution likelihood. The other two are the *infection probability*, defined as the probability that the fault will corrupt the data state of the program, and the *propagation probability*, the likelihood that the corrupted data will propagate to output and as such be observable.

Some of the issues raised by software inspection tools, such as coding standard violations, do not suit the latter two concepts, however. Furthermore, infection and propagation are in a way characterized by the severity level usually reported by such tools. Finally, while in [13] the metrics are obtained through dynamic analysis, we opt for static analysis, to be able to deploy the tool even before integration and testing, and integrate it seamlessly with existing automatic code inspection tools.

Evaluation Evaluation was done by means of a number case studies, including part of the software embedded in a television set, provided by our industrial partners. Although an in-depth discussion of the results is considered out of scope for this paper, we briefly remark on two important criteria; scalability and accuracy. The first aim was to produce an approach which would scale well, and we accomplished this by using simple graph traversals on a dependence graph representation (SDG) [7] of the source code. This produced an approach which scaled linearly with the program size, a relation we validated by testing the analysis on several programs of increasing size. Accuracy was evaluated by comparing dynamically obtained values for the execution likelihood to the estimates as produced by the analysis, and this delivered promising results. However, a number of issues remain:

- We need a better understanding of the accuracy/performance tradeoff. This can be accomplished by extending the approach with more sophisticated analyses that can produce more reliable estimates, and measuring the extra overhead involved in performing them.
- Though we have some idea of the accuracy of the analysis, we still need more information on how it works in practice. For this we intend to deploy the tool with our industrial partners and gain valuable developer feedback.

These issues, together with the proposed solution directions from the previous section, provide a venture point for future work.

4. Concluding Remarks

In this paper, we have established the necessity for research to support the reliable evolution of software, and discussed the context in which we plan to do this work. We then described a number of different directions in which solutions may be pursued by means of a set of research question, accompanied by a sketch of what these solutions will look like. In this section we will conclude by discussing validation of research results and the expected contributions.

[1] www.programmingresearch.com
[2] www.grammatech.com

Validation Our research questions were derived from discussions with our industrial project partners, which underlines their practical importance. Moreover, industry can provide an excellent platform for evaluation of the solutions we will develop. In other words: apart from tackling relevant problems, we stress the importance of a thorough validation of the resulting approach. One obvious important criterion here is *accuracy*: we need analyses that produce results that are sufficiently accurate to make their application feasible. Moreover, as we want to be able to apply our solutions to large, real-world industrial systems, we should evaluate their *scalability*. It has been remarked before [2] that small applications do not exhibit evolution-related problems, so our techniques must scale up to be useful. The techniques will be tested on a number of different open-source systems to evaluate with regard to both criteria. An example of such an evaluation has been given in Section 3. In addition, we will apply them to the systems developed within industry, and seek developer feedback on both their use in practice, and the quality of the results they produce.

Expected contributions Succesful solutions to the problems represented by the research questions will help developers to manage the reliability aspect of their software's evolution. Specifically, in the next paragraphs the contributions, together with potential pittfals, are discussed per question.

The major contribution in answering RQ1 would be to use fault tracking databases to empirically establish a correlation between actually observed issues and certain suspect code patterns. This can greatly enhance our ability to assess the predictive quality of defect detection techniques, and indicate areas where improvement is needed. Foreseen pitfalls are that the information contained within these databases may not be sufficiently detailed to support these kind of conclusions, and that there may not be enough issues that can actually be translated to abstract, generic code patterns.

A solution to the problem represented by RQ2 would leverage existing software inspection tools, increasing their usability and speeding their adoption. It would enable developers faced with severe time limits to pick out and solve those issues that have the greatest impact first. An important potential difficulty is to relate system-level notions of relevance or impact to the kind of source-code level metrics produced by source code analyses.

In our proposed solution direction for RQ3, we create a more robust handling of exceptions, increasing the reliability. Moreover, we aid the developer in comprehension of exceptions by repairing defunct handling and propagation mechanisms. Finally, by adding an explicit handling mechanism to the source code, the system becomes more evolvable. The major foreseen obstacle on this road is code exhibiting a great degree of inconsistency with regard to exception handling. For instance, there may be a number of idioms in use for handling exceptions, but when they are applied inconsistently, this can result in having insufficient information to reconstruct a handling model.

Acknowledgements

I would like to thank Leon Moonen and Arie van Deursen for their guidance, ideas and constructive comments on my research. In addition, I thank Andy Zaidman and Peter Zoeteweij for providing feedback on a draft version of this paper.

This work has been carried out as part of the Trader project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs.

References

[1] IEEE Standard Glossary of Software Engineering Terminology. In *IEEE Std 610.12-1990*, 1990.

[2] Keith H. Bennett and Vaclav Rajlich. Software Maintenance and Evolution: a Roadmap. In *ICSE - Future of SE Track*, pages 73–87, 2000.

[3] C. Boogerd and L. Moonen. Prioritizing Software Inspection Results using Static Profiling. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 149–158. IEEE, 2006.

[4] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Syst. Jour.*, 15(3):182–211, 1976.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[7] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[8] Embedded Systems Institute. Trader project website. `http://www.esi.nl/trader/`.

[9] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *10th Intnl. Symposium on Static Analysis (SAS 2003)*, volume 2694 of *LNCS*, pages 295–315. Springer, 2003.

[10] B.P. Lientz and E.B. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[11] A. A. Porter, H. Siy, A. Mockus, and L.G. Votta. Understanding the sources of variation in software inspections. *ACM Trans. Softw. Eng. Meth.*, 7(1):41–79, 1998.

[12] G. W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Softw.*, 8(1):25–31, 1991.

[13] J.M. Voas and K.W. Miller. Software Testability: The New Verification. *IEEE Softw.*, pages 17–28, 1995.

[14] D. A. Wheeler, B. Brykczynski, and Jr. R. N. Meeson, editors. *Software Inspection : An Industry Best Practice*. IEEE, 1996.

[15] C. Wohlin, A. Aurum, H. Petersson, F. Shull, and M. Ciolkowski. Software Inspection Benchmarking - A Qualitative and Quantitative Comparative Opportunity. In *8th Intnl. Software Metrics Symposium (METRICS 2002)*. IEEE, 2002.

[16] S.S. Yau, J.S. Collofello, and T.M. MacGregor. Ripple Effect Analysis of Software Maintenance. In *Proceedings of the Second IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 60–65. IEEE, 1978.