This article is part of the IEEE Reliability Society 2009 Annual Technology Report.

A Research Agenda for Software Reliability

Arbi Ghazarian
Email: arbi@cs.toronto.edu

It is well known that software development organizations spend a sizeable amount of their budget to rectify the defects introduced into their software systems during the development process. In addition to increasing the development cost, software faults also hurt organizations consuming software products in terms of lost business, and dissatisfied customers. In the case of safety-critical applications, software faults can have catastrophic results. With the ever-increasing dependence of our civil infrastructure to the correct functioning of software systems, the need for methods for engineering reliable software systems grows rapidly. My position is that research in the area of software reliability should pursue three complementary strategies:

- defect prevention, identification of defect introduction mechanisms;
- defect mitigation, development of methodologies for building fault tolerant systems; and
- defect removal, provisioning of effective, affordable requirement-component traceability.

In following the defect prevention strategy, the aim of the reliability research should be to gain an in-depth understanding of the mechanisms that give rise to defects in software systems. Such an understanding can then be leveraged to devise tools, techniques, and processes that enable software developers to proactively counteract the identified defect introduction mechanisms. Because it is not clear to what extent results from studies of software defects in one domain or application type are applicable to another domain or application type, it is necessary to take a domain-specific approach, independently targeting specific domains such as Enterprise Information Systems (EIS), embedded systems, scientific software, etc. Case studies of software defects in real-world systems are a systematic empirical research method that can provide us with invaluable insights about defect introduction mechanisms. Conducting multitudes of case studies in various domains will allow us not only to characterize defects within each domain, but also to discover patterns of defects within and across software domains. In accordance with this line of research, we have conducted industrial case studies of defects on commercial software systems [3], [4].

Research in the area of defect mitigation, on the other hand, should focus on the development of constructive approaches to building fault tolerant software systems. A fruitful avenue of research in this area is to study, identify, and document design choices or alternatives that reduce defect rate during the construction of software systems. Such design choices can be represented as a catalogue of design rules or patterns. A methodology for building fault tolerant systems will then provide guidelines on the systemic application of the design rules to achieve fault tolerance during the construction of software systems.

Defect prevention, and defect mitigation strategies will help to reduce the rate of defects in software systems. However, in general, the complete elimination of software defects is almost impossible. Therefore, the defect removal research strategy mentioned above emphasizes traceability as a mechanism that allows developers to efficiently locate faulty code, perform change impact analysis, and rectify defects more reliably, in less time. Moreover, traceability of software requirements to source code elements increases the reliability of software systems by making it possible to verify that all the stated requirements are implemented in the system's source code, and that there is no unnecessary code in the system; that is, all source code elements can be traced back to their higher level requirements. Unfortunately, achieving traceability in software systems can be prohibitively expensive. In the sections that follow, we discuss the traceability problem, and propose research directions that can potentially lead to achieving affordable traceability.

The Traceability problem

One of the major problems with current widely-used opportunistic (i.e., experienced-based) software development approaches is that the artifacts created during the development process are not readily traceable to their predecessor and successor artifacts. This situation is known as the traceability problem. The lack of traceability, or insufficient traceability reduces the reliability of software systems. Over the past few decades, numerous approaches have been developed to detect and establish trace links between software artifacts. In spite of these efforts, traceability is still not regarded as a feasible option for most commercial software projects. High cost is among the major barriers to the adoption of these traceability approaches.

The Costs of Traceability

Conventional approaches to traceability involve creating and updating traceability matrices that establish trace links between different artifacts. Unfortunately, this approach to traceability incurs high costs, making it an infeasible option for most commercial software projects. Large systems are composed of thousands of interconnected components that together satisfy a large set of software requirements. The magnitude of work involved in manually finding the trace links between requirements and their corresponding components is immense. While some automation exists, capturing traces remains a largely manual process [2]. It requires a significant effort even for moderately complex systems [6].

Trace links, once established, should be maintained so as to consistently represent the current state of the relationships between various system process artifacts. Separate evolution of models and systems makes the trace links obsolete, hence they are mistrusted [5]. Artifacts being traced are constantly changing as the system is developed. Therefore, maintaining a traceability scheme is challenging [1], [7], [8], [9]. Outdated documentation has always been a problem in software projects. Establishing traceability involves creating a set of additional documents such as the traceability matrices, which adds up to the burden of keeping software artifacts updated. Because of these costs, traceability is only practiced in cases where it is required by contractual obligations, or in projects such as safety critical software systems where quality is more important than productivity or cost minimization.

Characterizing Conventional Approaches to Traceability

To understand the root causes of the problems in conventional traceability approaches, and to obtain insights into potential solutions, we performed an analysis of conventional traceability approaches. As a result of this analysis, we have identified a number of problematic characteristics that are common among these approaches. The following subsections consider these problems.

Problem 1: Reliance on External Traceability

An analysis of the nature of the problems reported in the literature for capturing trace information reveals that conventional traceability approaches provide *external traceability* (as opposed to *internal traceability*). The form of traceability offered by these approaches is considered external in the sense that traceability is not an internal characteristic or property of the systems' design and source code. Instead, traceability is achieved outside the design and the source code by creating external maps of the system, such as the traceability matrices that attempt to capture traces between traceable entities in the system. In other words, the design and the source code of the system are not traceable in their own rights. It is precisely the inefficiency inherent in the process of discovering existing links between traceable entities in various software artifacts that makes conventional approaches to traceability an infeasible approach in real-life situations where software systems are to incorporate thousands of components to satisfy a large number of requirements.

Problem 2: Capturing Low-Level Trace links

Another characteristic of current approaches to traceability is that they rely on capturing low-level trace information. For instance, these approaches attempt to capture trace links between individual requirements and concrete source code constructs such as classes or methods that satisfy these

requirements. As a result, they have to deal with large amounts of trace information primarily because, in a large scale software system with thousands of requirements and components that satisfy these requirements, there are too many entities that need to be traced. Managing (i.e., detecting, documenting, validating, and updating) this large volume of trace information poses a great challenge for conventional traceability approaches.

Problem 3: Separate Capture of Trace Information

In conventional approaches to traceability, trace information is not generated as a byproduct of following the normal development process. Therefore, a separate traceability process is needed to capture trace links between software artifacts created during the development process. The traceability process can be either proactive, or after-the-fact. In proactive approaches to traceability, the development process, and the traceability process run in parallel; as the development process produces artifacts, the traceability process establishes and manages links between these artifacts. In practice, the steps of the traceability process are integrated into the various stages of the development process. A drawback of this approach is that it makes the development process more intense. In after-the-fact approaches to traceability, the development process, and the traceability process run sequentially. This means that the steps of the traceability process are postponed to the end of the development process. The drawback of this approach is that, at this point in the development process, it is much harder to recover the lost trace information. The additional steps required for capturing trace information in conventional approaches (both proactive, and after-the-fact approaches) put heavy burden on project resources.

Criteria for an Acceptable Solution

Based on our discussion of the problematic characteristics of conventional traceability approaches in the previous section, we suggest that an efficient, affordable approach to the traceability problem should address the three above-mentioned problem areas. Therefore, an acceptable solution to the problem should exhibit three properties:

- reliance on internal traceability,
- capturing higher-level traceability information, and
- generation of trace information as a byproduct of following the usual development process.

These criteria for an acceptable solution to the traceability problem can be used as a research framework that guides research in the area of traceability.

References

[1] Cleland-Huand, J., Chang, C.K., and Christensen, M., “Event-Based Traceability for Managing Evolutionary Change,” IEEE Transactions on Software Engineering, Vol. 29, No. 9, pp. 1226-1242, Sep.2003.

[2] Egyed, A., Grunbacher, P., Heindl, M., and Biffl, S., “Value-Based Requirements Traceability: Lessons Learned,” Proceedings of the 15th International Requirements Engineering Conference - RE'07, pp. 115-118, 2007.

[3] Ghazarian, A., “A Case Study of Defect Introduction Mechanisms,” Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE 2009), Springer, Lecture Notes in Computer Science (LNCS), pp. 156-170, Amsterdam, The Netherlands, June 2009.

[4] Ghazarian, A., “A Case Study of Source Code Evolution,” Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), IEEE Computer Society, pp. 159-168, Kaiserslautern, Germany, March 2009.

[5] Murphy, G. C., Notkin, D., and Sullivan, K., “Software Reflextion Models: Bridging the Gap Between Source and High-Level Models,” Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 18-28, Oct. 1995.

[6] Ramesh, B., Powers, T., Stubbs, C., and Edwards, M., "Lessons Learned From Implementing Requirements Traceability," Crosstalk, 8(4), pp. 11-15, 1995.
[7] Strens, M. R., and Sugden, R. C., "Change Analysis: A Step Towards Meeting the Challenge of Changing Requirements," Proceedings of IEEE Symposium and Workshop on Engineering of Computer Based Systems, March 1996.
[8] Sugden, R. C., and Strens, M. R., "Strategies, Tactics, and Methods for Handling Change," Proceedings of IEEE Symposium and Workshop on Engineering of Computer Based Systems, pp. 457-462, March 1996.
[9] Zowghi, D., and Offen, R., "A logical Framework for Modeling and Reasoning about the Evolution of Requirements," Proceedings of Third IEEE Symposium on Requirements Engineering, Jan. 1997.