Understanding the Redundancy of Software Systems

Andrea Mattavelli
University of Lugano - Faculty of Informatics
via G. Buffi 13, 6904 Lugano, Switzerland
andrea.mattavelli@usi.ch - http://www.inf.usi.ch/phd/mattavelli

ABSTRACT

Our research aims to study and characterize the redundancy of software systems. Intuitively, a software is redundant when it can perform the same functionality in different ways. Researches have successfully defined several techniques that exploit various form of redundancy, for example for tolerating failures at runtime and for testing purposes.

We aim to formalize and study the redundancy of software systems in general. In particular, we are interested in the intrinsic redundancy of software systems, that is a form of undocumented redundancy present in software systems as consequence of various design and implementation decisions. In this thesis we will formalize the intuitive notion of redundancy. On the basis of such formalization, we will investigate the pervasiveness and the fundamental characteristics of the intrinsic redundancy of software systems. We will study the nature, the origin, and various forms of such redundancy. We will also develop techniques to automatically identify the intrinsic redundancy of software systems.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms

Measurement, Reliability

Keywords

Redundancy, equivalence, execution diversity

1. RESEARCH PROBLEM

Redundancy is often exploited to improve the reliability of a system through the replication of critical components or functionalities.

Our work focuses on software redundancy. Informally, a software system is redundant when it can perform the same logical computation through two or more executions that are at least partially different, either because the code being executed is different or because it is used in different ways.

Redundancy has been extensively applied to software systems for testing purposes [4, 7, 19] and for tolerating failures at runtime [1, 2, 14].

Traditional hardware fault tolerance techniques improve reliability by means of the replication of components. However, software systems suffer mostly from design and development faults that cannot be masked by simply replicating components. For this reason, the majority of the proposed techniques overcome such deterministic failures by deliberately designing and implementing multiple versions of the same components independently [1, 12, 14].

Redundancy has been exploited for software testing as well, in particular to deal with the oracle problem. To generate oracles from redundancy, researchers have proposed different approaches, from the design of independently written versions of the program, which are then used to compare the outcome [19], to oracles based on formal specifications such as assertions and algebraic specifications [4, 7].

More recently, a technique called automatic workarounds was shown to be effective in improving the reliability of software systems by exploiting a form of redundancy that is intrinsically present in software [2, 3]. This form of redundancy is not introduced deliberately, but it rather occurs as an *intrinsic phenomenon* in the development of software. As an example, consider the redundancy of the Java containers, where one may add an element using the *add* or the *addAll* method. This redundancy was not designed deliberately to improve reliability. Still, under certain conditions, one may use one method if the other one fails.

Researchers have used redundancy for various specific purposes. Our aim is to formalize and study software redundancy *more generally*. Our intuition is that a systematic and formal characterization of redundancy, independent of its applications, would lead to more (and more effective) uses of this redundancy.

The main expected contribution of this thesis will be the foundations for the study and the systematic use of the intrinsic redundancy of software systems. We will characterize redundancy through qualitative and quantitative models. In essence, these models will somehow express an equivalence in functionality in combination with a difference in execution. We will then use these models to study the nature, origins, and manifestations of the intrinsic redundancy of software, for example by correlating its pervasiveness within a system with particular features of its design or development process.

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

We will then show that such models are helpful directly, as they allow for a more effective exploitation of software redundancy, but also indirectly, as they may document important design decisions. Intrinsic redundancy stems from various sources, and it is not documented as such. Therefore, its identification requires a non-trivial manual analysis. We will further exploit both the formalization of redundancy and the quantitative and qualitative models to develop an approach to automatically identify intrinsic redundancy.

2. BACKGROUND AND RELATED WORK

There are several studies on the occurrence of some forms of redundancy in software. The largest body of work in this area focuses on the study of code clones. A code clone is a segment of program code that is syntactically identical or very similar to other segments in the same program. Most research work in this area focuses on the problem of detecting clones, especially in large code bases [9, 11]. Other studies have also looked at the evolution of clones and how to keep track and manage code clones [13].

Recent studies indicate that software is also semantically redundant. Gabel et al. used a graph-based approach to identify semantic code clones [5]. Their analysis of the Linux kernel found more than 3,000 semantic clones. Jiang and Su also looked for functionally equivalent code fragments that, given the same input, would compute the same output [10]. The study of Jiang and Su, which was also applied to the Linux kernel, reveals more than 600,000 semantically equivalent code fragments. Carzaniga et al. analyzed Javascript Web applications including Google Maps, Youtube, and JQuery, and identified more than 150 sequences of methods that replicate the functionalities of other methods.

The intrinsic redundancy of software was successfully exploited to test software and to build self-healing systems. In the context of service-oriented applications, the availability of multiple versions of similar services has been exploited to overcome failures caused by malfunctioning services or by not anticipated changes in the functionalities offered by the current implementation [16, 17]. Intrinsic redundancy was also used to avoid failures at runtime. Carzaniga et al. developed techniques for Javascript and Java applications that can avoid failures by replacing faulty sequences of operations with others that produce the same outputs or effects [2, 3].

Intrinsic redundancy has been exploited in software testing as well. Doong and Frankl proposed to exploit intrinsic redundancy to generate tests and oracles [4]. Their approach relies on algebraic specifications to identify equivalent sequences of operations in the form of axioms. The technique generates test cases that execute pairs of supposedly equivalent sequences and then compare the resulting objects using an equivalence check provided by the testers. In a similar way, Gotlieb considers operation permutations as alternative operations [7].

3. RESEARCH GOAL AND CHALLENGES

We propose a systematic study of the intrinsic redundancy of software systems. We will study the pervasiveness, the nature, and the characteristics of the intrinsic redundancy of modern software systems. This characterization will correlate the presence of this form of redundancy with some features of the design process and other relevant aspects of the development process of modern software systems.

We explore the following main research questions:

Q1 What is the essence of software redundancy?

Q2 How pervasive is the intrinsic redundancy in software systems?

Q3 Why is software intrinsically redundant? That is, what are the sources of intrinsic redundancy?

Q4 Is it possible to identify redundancy automatically?

There does not seem to be an agreed-upon and general definition of software redundancy [5, 10, 12, 18]. Thus the goal of **Q1** is to provide a formalization of the concept of software redundancy. Intuitively, redundancy amounts to performing the same function through different computations. So the main challenge is to define a framework capable of expressing the functional *equivalence* of two executions, and at the same time characterizing their *difference*. In particular, we are interested in a notion of equivalence that abstracts from the internal state representation, and yet characterizes quantitatively the difference between two executions in terms of the intermediate and low-level state transitions.

We propose to employ the formalized notion of redundancy to measure the pervasiveness of redundancy—and in particular intrinsic redundancy—in modern software systems (**Q2**). We plan to start form the redundancy that manifests itself at the method level. Then, we will move both to lower and higher levels, considering blocks and statements, as well as components and applications. The main challenge here is in the evaluation plan, and in particular in the selection of the subjects, which must be numerous and diverse enough to allow us to derive statistically significant results on the pervasiveness of intrinsic redundancy.

The study on the pervasiveness of redundancy in various software systems is fundamental to answer question **Q3**. In this research question our main goal is both to understand and to characterize the nature of intrinsic redundancy. We will determine if, and to what extent, aspects of the software development process affect, and are affected, by the presence of intrinsic redundancy. We will investigate whether specific technical aspects, such as the choice of the design paradigm or the need to guarantee backward compatibility, induce more or less intrinsic redundancy in software systems. Our intuition is that reusable libraries are more intrinsically redundant simply because they are designed to offer the same functionality through different interfaces, and that diversity of interfaces leads quite naturally to redundancy. Similarly, the need to guarantee backward compatibility may increase the intrinsic redundancy of software systems.

To answer question **Q4** we aim to define techniques to automatically identify the intrinsic redundancy of software systems. This has been recognized as an open and challenging problem [18]. We plan to define both a technique to identify equivalent sequences of methods, and a technique to quantitatively measure the differences in execution between two equivalent sequences of methods. We plan to develop a search-based algorithm to synthesize sequences of methods whose behavior is equivalent to a given method. Then, with a set of equivalent sequences of methods, we can exploit the quantitative metrics proposed to answer question **Q1** to characterize the difference between two executions.

Table 1: Equivalent method sequences found in representative Java libraries and applications

Library	Ant	Guava	SWT	JodaTime	Lucene	GraphStream
Classes considered	213	118	252	12	160	9
Equivalences found	804	2078	1494	135	205	132
Average per class	3.8	17.6	5.93	11.25	1.28	14.67

4. RESEARCH STATUS

In the first part of the PhD we focused mostly on formalizing the notion of redundancy, and on understanding the pervasiveness of intrinsic redundancy. Beside those tasks, we have also tried both to formulate hypotheses on the nature and the origin of intrinsic redundancy, and to define an approach to automatically identify intrinsic redundancy.

4.1 Formalizing Redundancy

The first concept we express within our framework is equivalence. In particular, we are interested in functional equivalence with respect to the *intended* behavior of the system as defined in the specifications. We express the notion of functional equivalence by defining a relation between two different sequences of method calls M and M'. We refer to the method sequences involved in such kind of relation as *equivalent sequences*. Equivalent sequences are defined as alternative sequences of operations whose intended behavior is equivalent to that of the original sequence [2, 3].

We must then develop this notion of functional equivalence, and in particular we intend to adopt a notion of equivalence based on observable behavior rather than on internal state representation. For this reason, we base our definition on the notion of observational equivalence [8]. We consider equivalent two methods sequences M and M' if their execution can not be distinguished. In particular, the execution of M and M' on two objects may leave the objects in two different internal states. However, if those two states cannot be distinguished through any (potentially infinite) sequence of methods of the public interface of the object, then we say that the two sequences are equivalent.

The second concept to express within our framework is a characterization of the differences between two executions. In particular, we are interested in quantitative metrics of such differences to measure the amount of redundancy between equivalent sequences. For this purpose, we are investigating various distance metrics based on dynamic analysis such as execution traces. The idea is to extract traces from the execution of both the original and the equivalent sequence, and to compare the two traces to detect any difference in the behavior of the program. To quantify the differences between traces we are experimentally evaluating various distance function, including some inspired by DNA alignment algorithms [6] and others used for fault localization [15].

4.2 Observing Intrinsic Redundancy

To characterize intrinsic redundancy, we have first conducted an empirical study to quantify the presence of semantically equivalent sequences of methods, especially in reusable libraries. We systematically studied and documented the equivalence of some Java libraries and applications of non-trivial size and complexity, such as Apache Ant, Apache Lucene, Google Guava, Joda-Time, Eclipse SWT, and GraphStream. We identified equivalences between sequences of library calls by first reading the documentation and later testing experimentally each pairs of sequences for equivalence. The results of our investigation are summarized in Table 1.

The notion of redundancy combines equivalence in functionality with diversity in execution. Therefore, once we identified equivalent sequences of methods, we also tried to determine to what extent their executions differ. We are still investigating which distance measure is more suitable for this purpose, and therefore we do not yet have conclusive quantitative measures of diversity of executions and therefore of redundancy. However, both a manual inspection and our previous work suggest that a significant amount of equivalence indeed corresponds to redundant behaviors [2].

4.3 Characterizing Intrinsic Redundancy

We plan to proceed experimentally bottom-up by first formulating hypotheses about the source of intrinsic redundancy, and then validating these hypotheses by correlating the presence of redundancy with one of the hypothesized sources. During our empirical investigation we observed plausible and general reasons that justify the presence of intrinsic redundancy in modern software systems, especially in modular and reusable components.

Modern development practices naturally induce developers to use reusable components that already implement the needed functionality. It is common to find several components that provide similar or identical functionalities. For example, the Trove4j library implements collections specialized for primitive types and overlaps with the standard Java library. Other similar occurrences of replicated functionalities can be found in the Apache Ant project.

Performance optimization and context-specific optimization are another source of redundancy. For example, the GNU Standard C++ Library implements its basic stable sorting function using the insertion-sort algorithm for small sequences, and merge-sort for the general case.

Yet another source of redundancy is due to the need to guarantee backward compatibility. A library might maintain different versions of the same components to ensure compatibility with previous versions. For example, the Java 7 standard library contains at least 45 classes and 350 methods that are deprecated and that overlap with the functionality of newer classes and methods.

Finally, even when and in fact *because* software is extensively modularized, redundancy occurs as a by-product of the design of highly configurable and versatile libraries or frameworks.

4.4 Synthesizing Equivalent Sequences

So far, we have manually identified equivalent sequences by first identifying possible candidates and then testing the sequences for equivalence. We are now investigating a search-based technique to detect intrinsic redundancy automatically.

To automatically generate an equivalent sequence, we need to find a sequence of objects and method calls that returns the same result as the original one and leads to a state

that is indistinguishable through its observable behavior. We are studying genetic algorithms to synthesize equivalent sequences. In particular, we are considering a two-stage approach that first generates candidate equivalent sequences, and then refines the result by removing spurious sequences that can be shown not to be equivalent.

The approach starts with a target method for which we want to find an equivalent sequence. In the first phase, we generate a set of usage scenarios for the target method. We then exploit a genetic algorithm to synthesize sequences of method calls that behave as the target method on all usage scenarios. In the second stage, we remove spurious results, that is candidate sequences whose behavior diverges from the original one. We exploit a genetic algorithm to generate counterexamples that invalidate the equivalence between the original and the candidate behaviors. If the algorithm can not synthesize a counterexample within a fixed amount of time, we consider that candidate as a valid equivalent sequence.

We performed a preliminary evaluation to prove the feasibility of the approach. We selected few simple examples, as the *Stack* class of the Java standard library. The approach succeeded in generating equivalent sequences for methods such as *pop()*, *push()* and *remove()*.

5. EVALUATION PLAN

We plan to evaluate our study on software redundancy through an extensive and systematic empirical evaluation. The objective of the evaluation is to validate the proposed formalization, the characterization, and the automatic identification of intrinsic redundancy.

First we plan to extend our preliminary study on the pervasiveness of intrinsic redundancy at the method level. We plan to investigate subjects of different size, complexity, application domains, design paradigms, and programming languages. We then plan to evaluate our characterization of intrinsic redundancy by correlating the presence of intrinsic redundancy with our hypotheses on the nature and the origin of intrinsic redundancy. Again, our goal is to determine if and to what extent these aspects induce various forms of redundancy.

We plan to investigate the accuracy of the framework to express sequences of methods that can be considered redundant by comparing the equivalent sequences identified according to our definition with an intuitive notion of redundancy based on human independent judgment.

We plan to evaluate the effectiveness of the automatic identification of intrinsic redundancy through a comparative study that compares the equivalent sequences manually identified with the ones automatically identified. We will take into consideration several applications and libraries to reduce possible bias due to either technical or environmental aspects of the software systems.

6. REFERENCES

[1] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[2] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *Proc. of the International Conference on Software Engineering*, pages 782–791, 2013.

[3] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for Web applications. In *Proc. of the 2010 Foundations of Software Engineering Conference*, pages 237–246, 2010.

[4] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. on Soft. Engineering and Methodology*, 3(2):101–130, 1994.

[5] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. of the Intl. Conference on Software Engineering*, pages 321–330, 2008.

[6] D. Gao, M. K. Reiter, and D. Song. Behavioral distance for intrusion detection. In *Proc. of the International Conference on Recent Advances in Intrusion Detection*, pages 63–81, 2006.

[7] A. Gotlieb. Exploiting symmetries to test programs. In *Proc. of the International Symposium on Software Reliability Engineering*, pages 365–374, 2003.

[8] M. Hennessy and R. Milner. On observing non determinism and concurrency. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85, pages 299–309. Springer, 1980.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. of the International Conference on Software Engineering*, pages 96–105, 2007.

[10] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 81–92, 2009.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[12] J.-C. Laprie, C. Béounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, July 1990.

[13] J. R. Pate, R. Tairas, and N. A. Kraft. Clone evolution: a systematic review. *Journal of Software: Evolution and Process*, 25(3):261–283, 2013.

[14] B. Randell. System structure for software fault tolerance. *SIGPLAN Notes*, 10(6):437–449, 1975.

[15] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. of the International Conference on Automated Software Engineering*, pages 30–39, 2003.

[16] S. Subramanian, P. Thiran, N. C. Narendra, G. K. Mostefaoui, and Z. Maamar. On the enhancement of BPEL engines for self-healing composite web services. In *Proc. of the International Symposium on Applications and the Internet*, pages 33–39, 2008.

[17] Y. Taher, D. Benslimane, M.-C. Fauvet, and Z. Maamar. Towards an approach for Web services substitution. In *Proc. of the Intl. Database Engineering and Applications Symposium*, pages 166–173, 2006.

[18] A. Walenstein, M. El-Ramly, J. R. Cordy, W. S. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, and J. W. von Gudenberg. Similarity in programs. In *Dagstuhl Seminar Proceedings*, 2007.

[19] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.