

Preface

The idea of improving software quality through reuse is not new. After all, if software works and is needed, just reuse it. What is new and evolving is the idea of relative validation through testing and reuse and the abstraction of code into frameworks for instantiation and reuse. Literal code can be abstracted. These abstractions can be made to yield similar codes, which serve to verify their patterns. There is a taxonomy of representations from the lowest level literal codes to their highest level natural language descriptions. The quality of software is improved in proportion to the degree of all levels of reuse.

Any software, which in theory is complex enough to allow for self-reference, cannot be assured to be absolutely valid. The best that can be attained is a relative validity, which is based on testing. Axiomatic, denotational, and other program semantics are more difficult to verify than the codes, which they represent! But, is there any limit to testing, and how does one maximize the reliability of software through testing? These are the questions, which need to be asked. Here are the much sought-after answers.

Randomization theory implies that software should be broken into small coherent modules, which are logically integrated into the whole. These modules are designed to facilitate specification and may be optimized through the use of equivalence transformations. Moreover, symmetric testing (e.g., testing a sort routine using (3, 2, 1) (4, 3, 2, 1) (5, 4, 3, 2, 1)...) has minimal value beyond the first test case. Rather, the test cases need to cover all combinations and execution paths, which defines random-basis testing (e.g., (3, 2, 1) (3, 1, 2) (1) (2, 1, 2)...).

It is virtually impossible to generate a complete random-basis test for complex software. Hence, reuse in diverse applications serves as a substitute. Also, the more coherent the software modules, the greater their potential for reuse and integration, and the proportionately less the propensity for software bugs to survive as a consequence. Such an approach implies not only the use of integration testing, but the definition and application of true optimizing compilers as well.

For example, routines for reading data, iteratively bubbling the data to the list head to create a lexicographic order, and printing the result of the iterations (e.g., the $O(n^2)$ bubble sort) can be iteratively transformed, by an optimizing compiler,

into an $O(n \log n)$ sort (e.g., Quicksort). Such coherent optimizations may be self-referentially applied to the optimizing compilers themselves for further efficiencies of scale.

The aforementioned ways for optimizing literal software quality extend to all of its abstract patterns—only here testing applies to framework instances. It can be seen that software quality is properly dependent upon the scale of the effort. Scale, in turn, is defined by the embodied knowledge (i.e., including frameworks) and processor power, which can be brought to bear. Randomization theory and applicative experience converge as software is written for coherency, integrated for functionality, optimized for efficiency, executed to realize its latent potential for massive parallelism, and reused in diverse applications to maximize its validity (i.e., through random-basis testing).

One final note is appropriate. It should be possible to create domain-general higher-level (i.e., context-sensitive) languages through the inclusion of dialog, for disambiguation, and the capture of knowledge in the form of software frameworks. The higher the level of software, the more reusable it is on account of one central human factor—readability. That is why components are reusable—because their descriptive characterizations are understandable. This also follows from randomization theory although this theory does not address how to achieve such context-sensitive languages in practice—only the result of doing so.

Combining the key points herein leads to the inescapable conclusion that software productivity—including quality attributes—is not bounded, combines the best of theory and practice, and when realized, as described, will transform the software industry, as we know it, for the better.

The traveling salesman problem (TSP) is one of the most studied problems in optimization. If the optimal solution to the TSP can be found in polynomial time, it would then follow that every *NP-hard* problem could be solved in polynomial time, proving $P=NP$. In Chapter 1, it will be shown that the proposed algorithm finds $P \sim NP$ with scale. Machine learning through self-randomization is demonstrated in the solution of the TSP. It is argued that self-randomizing knowledge bases will lead to the creation of a synthetic intelligence, which enables cyber-secure software automation. The work presented in Chapter 2 pertains to knowledge generalization based on randomization. Inductive knowledge is inferred through transmutation rules. A domain-specific approach is properly formalized to deal with the transmutation rules. Randomized knowledge is validated based on the domain user expertise.

In order to improve software component reuse, Chapter 3 provides a mechanism, based on context-sensitive code snippets, for retrieving components and showing how the retrieved components can be instantiated and reused. This approach utilizes semantic modeling and ontology formalisms in order to conceptualize and reverse engineer the hidden knowledge in library codes. Ontologies are, indeed, a standard in representing and sharing knowledge. Chapter 4 presents a comprehensive methodology for ontology integration and reuse based on various matching techniques. The approach is supported by an ad hoc software framework, whose scope

is easing the creation of new ontologies by promoting the reuse of existing ones and automating, as much as possible, the entire ontology construction procedure.

Given ever-increasing data storage requirements, many distinct classifiers have been proposed for different data types. However, the efficient integration of these classifiers remains a challenging research topic. In Chapter 5, a novel scalable framework is proposed for a classifier ensemble using a set of generated adjudications based on training and validation results. These adjudicators are ranked and assembled as a hierarchically structured decision model. Chapter 6 presents a graph-based solution for the integration of different data stores using a homogeneous representation. Schemas are transformed and merged over property graphs, providing a modular framework.

In chapter 7, heterogeneous terminologies are integrated into a category-theoretic model of faceted browsing. Faceted browsing systems are commonly found in online search engines and digital libraries. It is shown that existing terminologies and vocabularies can be reused as facets in a cohesive, interactive system. Controlled vocabularies or terminologies are often externally curated and are available as a reusable resource across systems. The compositional reuse of software libraries is important for productivity. However, the duplication and modification of software specifications, for their adaptation, leads to poor maintainability and technical debt. Chapter 8 proposes a system that solves these problems and enables the compositional reuse of software specifications—*independent of the choice of specification languages and tools*.

As the complexity of the world and human interaction grows, contracts are necessarily becoming more complex. A promising approach for addressing this ever-increasing complexity consists in the use of a language having a precise semantics—thus providing a formal basis for integrating precise methods into contracts. Chapter 9 outlines a formal language for defining general contracts, which may depend on temporally based conditions. Chapter 10 outlines the design of a system modeling language in order to provide a framework for prototyping complex distributed system protocols. It further highlights how its primitives are well-matched with concerns, which naturally arise during distributed system design. An operational semantics for the designed language, as well as results on a variety of state-space exploration techniques, is presented.

Chapter 11 addresses the integration of formalisms within the behavior-driven development tooling, which is built upon semi-formal mediums for specifying the behavior of a system as it would be externally observed. It presents a new strategy for test case generation. Chapter 12 focuses upon model checking Z specifications. A Z specification is preprocessed, where a generic constant is redefined as an equivalent axiom, and a schema calculus is expanded to a new schema definition. Chapter 13 presents a parameterized logic to express nominal and erroneous behaviors—including faults modeling, given algebra and a set of operational modes.

The created logic is intended to provide analysts with assistance in considering all possible situations, which may arise in complex expressions having order-related operators. It enables analysts to avoid missing subtle (but relevant) combinations.

January 2017

Stuart H. Rubin
Thouraya Bouabana-Tebibel



<http://www.springer.com/978-3-319-56156-1>

Quality Software Through Reuse and Integration

Rubin, S.H.; Bouabana-Tebibel, T. (Eds.)

2018, X, 313 p. 84 illus., Softcover

ISBN: 978-3-319-56156-1