AUTOMATED DATABASE APPLICATIONS TESTING

Specification Representation for Automated Reasoning

Rana Farid Mikhail
Donald Berndt
Abraham Kandel

SERIES IN
MACHINE PERCEPTION
ARTIFICIAL INTELLIGENCE
Volume 76

World Scientific

Series in Machine Perception and Artificial Intelligence – Vol. 76

AUTOMATED DATABASE APPLICATIONS TESTING

Specification Representation for Automated Reasoning

Rana Farid Mikhail, Donald Berndt & Abraham Kandel
University of South Florida, USA

World Scientific
NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI

Published by

World Scientific Publishing Co. Pte. Ltd.

5 Toh Tuck Link, Singapore 596224

USA office: 27 Warren Street, Suite 401-402, Hackensack, NJ 07601

UK office: 57 Shelton Street, Covent Garden, London WC2H 9HE

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

Series in Machine Perception and Artificial Intelligence — Vol. 76

AUTOMATED DATABASE APPLICATIONS TESTING

Specification Representation for Automated Reasoning

ISBN-13 978-981-283-728-8

ISBN-10 981-283-728-0

Printed in Singapore by B & Jo Enterprise Pte Ltd

AUTOMATED DATABASE APPLICATIONS TESTING

Specification Representation for Automated Reasoning

SERIES IN MACHINE PERCEPTION AND ARTIFICIAL INTELLIGENCE*

Editors: **H. Bunke** (Univ. Bern, Switzerland)
P. S. P. Wang (Northeastern Univ., USA)

Vol. 61: Decomposition Methodology for Knowledge Discovery and Data Mining: Theory and Applications
(*O. Maimon and L. Rokach*)

Vol. 62: Graph-Theoretic Techniques for Web Content Mining
(*A. Schenker, H. Bunke, M. Last and A. Kandel*)

Vol. 63: Computational Intelligence in Software Quality Assurance
(*S. Dick and A. Kandel*)

Vol. 64: The Dissimilarity Representation for Pattern Recognition: Foundations and Applications
(*Elżbieta Pękalska and Robert P. W. Duin*)

Vol. 65: Fighting Terror in Cyberspace
(*Eds. M. Last and A. Kandel*)

Vol. 66: Formal Models, Languages and Applications
(Eds. *K. G. Subramanian, K. Rangarajan and M. Mukund*)

Vol. 67: Image Pattern Recognition: Synthesis and Analysis in Biometrics
(Eds. *S. N. Yanushkevich, P. S. P. Wang, M. L. Gavrilova and S. N. Srihari*)

Vol. 68: Bridging the Gap Between Graph Edit Distance and Kernel Machines
(*M. Neuhaus and H. Bunke*)

Vol. 69: Data Mining with Decision Trees: Theory and Applications
(*L. Rokach and O. Maimon*)

Vol. 70: Personalization Techniques and Recommender Systems
(Eds. *G. Uchyigit and M. Ma*)

Vol. 71: Recognition of Whiteboard Notes: Online, Offline and Combination
(Eds. *H. Bunke and M. Liwicki*)

Vol. 72: Kernels for Structured Data
(*T Gärtner*)

Vol. 73: Progress in Computer Vision and Image Analysis
(Eds. *H. Bunke, J. J. Villanueva, G. Sánchez and X. Otazu*)

Vol. 74: Wavelet Theory Approach to Pattern Recognition (2nd Edition)
(*Y Y Tang*)

Vol. 75: Pattern Classification Using Ensemble Methods
(*L Rokach*)

Vol. 76 Automated Database Applications Testing: Specification Representation for Automated Reasoning
(*R F Mikhail, D Berndt and A Kandel*)

*For the complete list of titles in this series, please write to the Publisher.

Dedication Page

To my beloved father, Dr. Farid Halim Mikhail - All I am, I owe to you. Your words of wisdom will remain in my heart and mind as long as I live.

To my husband and best friend, Emad - You're the best part of my day - every day!

To my mother Nadia and brother Ramy, every day I appreciate you more and learn how blessed I am, to have you as my family.

Rana Mikhail

To my mother Theresa Ciaravino Berndt, who lived a life of faith and family.

Don Berndt

Dedicated to the memory of my dear brother Professor Shmuel Kandel.

Abraham Kandel

Preface

In this book we introduce SpecDB, a database created to represent and host software specifications in a machine-readable format. The specifications represented in SpecDB are for the purpose of unit testing database operations. A structured representation aids in the processes of both automated software testing and software code generation, based on the actual software specifications. We describe the design of SpecDB, the underlying database that can hold the specifications required for unit testing database operations.

Specifications can be fed directly into SpecDB, or, if available, the formal specifications can be translated to the SpecDB representation. An algorithm that translates formal specifications to the SpecDB representation is described. The Z formal specification language has been chosen as an example for the translation algorithm. The outcome of the translation algorithm is a set of machine-readable formal specifications.

To demonstrate the use of SpecDB, two automated tools are presented. The first automatically generates database constraints from represented business rules in SpecDB. This constraint generator gives the advantage of enforcing some business rules at the database level for better data quality. The second automated application of SpecDB is a reverse engineering tool that logs the actual execution of the program from the code. By Automatically comparing the output of this tool to the specifications in SpecDB, errors of commission are highlighted that might otherwise not be identified. Some errors of commission including coding unspecified behavior together with correct coding of the specifications cannot be discovered through black box testing techniques, since these techniques cannot observe what other modifications or outputs have happened in the background. For example, black box, functional testing techniques cannot identify an error if the soft-

ware being tested produced the correct specified output but more over, sent classified data to insecure locations. Accordingly, the decision of whether a software application passed a test depends on whether it coded all the specifications and only the specifications for that unit. Automated tools, using the reverse engineering application introduced in this book, can thus automatically make the decision whether the software passed a test or not based on the provided specifications.

This book is intended for to the members of the software testing team and developers of software testing tools. System analysts and designers could also benefit from the ideas presented in this book. Although the title of the book caters for database applications, yet, those interested in software applications that do not involve database systems can also benfit greatly from the concepts in this book, since applications based on database systems are a superset of those that do not access databases.

Acknowledgments

The authors would like to acknowledge the National Institute for Systems Test and Productivity (NISTP) at the University of South Florida for funding parts of this work and other related projects, under sponsorship from the USA Space and Naval Warfare Systems Command (N00039-01-1-2248). NISTP supported a multi-year effort focused on investigating the intersection of computational intelligence and software quality assurance. We have been fortunate to be part of a changing group of interdisciplinary researchers that were free to pursue a variety of somewhat unusual projects. We are thankful for the opportunity.

Contents

Chapter 1

Introduction

1.1 The Need for Testing

One of the common but very true jokes about the software industry tells of a software entrepreneur proudly stating that if the automobile industry had advanced in the past decade with the same rate as that of the computer industry, we would all be driving $25 cars that run for a 1,000 miles per gallon of gasoline. To which an automobile CEO replies, that if cars were like today's software, they would crash twice a day for no reason, and when you call for service, they would tell you to shut down the engine, as well as all windows, get out of the car, lock it, and start over! Indeed, in the past decade, computer hardware has advanced significantly offering much higher processing capability and speed. Consequently, the complexity and size of software has increased exponentially too. Software is vital to almost every industry and we are becoming ever more dependent on our computers to perform every day tasks. As the complexity and size of software grow, so does the need to insure the reliability and precision of software outputs. Software errors can be fatal in numerous industries; including airlines, medical and the military. Billions of dollars are spent on software and software testing and maintenance annually by the DoD. Software errors result in hundreds of millions in losses annually from the DoD alone [1] (software errors are defined in Section 1.3.1).

Accordingly, a lot of attention is now directed towards assuring software reliability, accuracy, security and usefulness. It is crucial to dedicate a lot of resources, including time, effort and a budget during the software lifecycle to verify that the end product is useful, usable and accomplishes the tasks it was built to perform.

1.2 Why Does Software have Errors?

Software errors result from many reasons, from misunderstanding of the requirements, to poor design; also from tired or careless programmers to rushed deadlines [2]. Often testing is delayed until later in the software development life cycle, and little time is given for testing, debugging, and re-testing. Coverage gaps are also a problem; this happens when not all parts of the software are tested.

Some software failures happen due to a specific scenario, a sequence of events that might have been tested individually but not in that particular sequence, that causes the software failures. The operating environment, various connected hardware peripherals, other concurrently running software and different operating systems could also be a source of software failures [4].

It was proven that complete testing is impossible [3]. Meaning that it is impossible to test all values of each possible input, valid and invalid, with their respective combinations for a software application with considerable size and complexity; the number of test cases will be astronomical and the time needed for testing and verification will be almost endless. For example, a very simple program might take only three inputs, an integer Age, between 0 and 99, a character, Initial, between A and Z, and another integer representing a Month between the integers 1 and 12.

```
Age: integer [0..99]
Initial: char [A..Z]
Month: integer [1..12]
```

For this simple program, there are 100 possible valid values for Age, 26 valid values for Initial, and 12 valid values for Month. Accordingly there are 100*26*12 = 31,200 possible combinations of valid values. If complete testing was required, there would be 31,200 test cases that should be carried out to test the operation of this program on valid values alone. Even if the 31,200 test cases were run, this would not be considered complete testing. The number of test cases required for complete testing will be much greater, when the invalid values are added. Invalid values could be in one, two or all three inputs. The number will increase exponentially if we want to create test cases for all these cases; where only one of the three variables contains an invalid value, a combination of two of the variables are invalid, or all three. For example, the Age variable is an integer, one

can enter any integer from the negative integer boundary to the positive integer boundary. Also, one can try to enter characters, real numbers or any other value for the Age integer value and test the program behavior and how it handles exceptions. The possibilities and combinations of all those cases for this very simple three input program are thus almost endless. Also it is impossible to completely test all values of real numbers, even in a small range, and for strings representing names for example, etc. Some transient errors are revealed in complex distributed or embedded control systems after prolonged usage. This may be a result of a memory overflow, or minute system clock discrepancies that accumulate to a noticeable difference after prolonged usage. Such errors are very difficult to recognize [77], and require high volume system testing or long sequence testing techniques, and extended random regression testing [79]. Hence there is a need for better software testing techniques, which focus on testing parts of the input domain, yet, attain good coverage to verify software quality.

1.3 Software Testing Definitions

1.3.1 *Software Errors, Faults and Failures*

A software error is a mismatch between the program and its specifications, provided that the specifications are accurate and complete. A software application is considered erroneous if it fails to be useful. A software fault is defined as the execution of an error [5]. Faults of commission are those resulting from incorrectly coding specifications, or coding unspecified requirements. Faults of omission result from failing to implement specified requirements. A software failure happens when a fault of commission is executed while running the software application [5]. From Figure 1.1, the goal of testing is to make sure that $S \cap P = S = P$. If the above equation is reached in testing a software application, the resulting product is very well tested and functional, provided that the specifications are correct, accurate, complete and do not contain any contradictions.

1.3.2 *Software Testing*

Software testing is becoming an important area of research aimed at producing more reliable systems and minimizing programming errors [78]. Since it is impossible to completely test how a software application performs on all values of all of its inputs, as described in Section 1.2, a good strategy to test

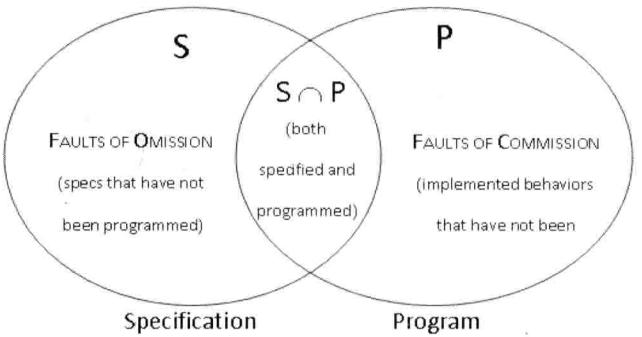

Fig. 1.1 The Relationship Between a Program Specification and its Implementation [5]

software has to be followed in order to choose test cases that test the software without coverage gaps. However, there is much more to good testing than running a program several times to check if it functions correctly [6]. In order to systematically test software, we need to model the environment in which the software is intended to run, select test scenarios that attain good system coverage, execute and evaluate the test cases, correct the errors, and measure the testing progress [4]. Some faults may result from correcting other errors, and hence regression testing is often used, which entails re-running all test cases after an error has been corrected to verify that the fix did not inject any errors in a part of the software that was previously running correctly. Software testing has been defined in many ways. Some authors define it as the process of running a software application to verify its correct execution of its valid and complete specifications, and verify that it runs correctly in the environment it was created for [5, 86]. Other authors argue that test scenarios that do not reveal errors or failures are also of great importance, since the testing team is assuring by such scenarios that the system actually implemented the required specifications correctly [6]. Software can also fail by executing too slowly or using too much memory; reasonable execution speed and memory usage are obvious requirements that might not be in the specification documents. A number of good references for software testing are [5, 6, 553]. Accordingly, it is crucial to create test cases that reflect a good coverage of all areas of the