VALID 2011

The Third International Conference on Advances in System Testing and Validation Lifecycle

ISBN: 978-1-61208-168-7

October 23-29, 2011

Barcelona, Spain

VALID 2011 Editors

Teemu Kanstrén, VTT Technical Research Centre of Finland - Oulu, Finland

VALID 2011

Forward

The Third International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011), held on October 23-29, 2011 in Barcelona, Spain, continued a series of events focusing on designing robust components and systems with testability for various features of behavior and interconnection.

Complex distributed systems with heterogeneous interconnections operating at different speeds and based on various nano- and micro-technologies raise serious problems of testing, diagnosing, and debugging. Despite current solutions, virtualization and abstraction for large scale systems provide less visibility for vulnerability discovery and resolution, and make testing tedious, sometimes unsuccessful, if not properly thought from the design phase.

The conference on advances in system testing and validation considered the concepts, methodologies, and solutions dealing with designing robust and available systems. Its target covered aspects related to debugging and defects, vulnerability discovery, diagnosis, and testing.

The conference provided a forum where researchers were able to present recent research results and new research problems and directions related to them. The conference sought contributions presenting novel result and future research in all aspects of robust design methodologies, vulnerability discovery and resolution, diagnosis, debugging, and testing.

We welcomed technical papers presenting research and practical results, position papers addressing the pros and cons of specific proposals, such as those being discussed in the standard forums or in industry consortiums, survey papers addressing the key problems and solutions on any of the above topics, short papers on work in progress, and panel proposals.

We take here the opportunity to warmly thank all the members of the VALID 2011 technical program committee as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and efforts to contribute to the VALID 2011. We truly believe that thanks to all these efforts, the final conference program consists of top quality contributions.

This event could also not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the VALID 2011 organizing committee for their help in handling the logistics and for their work that is making this professional meeting a success. We gratefully appreciate to the technical program committee co-chairs that contributed to identify the appropriate groups to submit contributions.

We hope the VALID 2011 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in system testing and validation.

We hope Barcelona provided a pleasant environment during the conference and everyone saved some time for exploring this beautiful city.

VALID 2011 Chairs

Advisory Chairs
Andrea Baruzzo, Università degli Studi di Udine, Italy
Cristina Seceleanu, Mälardalen University, Sweden
Mehdi Tahoori, Karlsruhe Institute of Technology (KIT), Germany
Mehmet Aksit, University of Twente - Enschede, The Netherlands
Amirhossein Alimohammad, Ukalta Engineering - Edmonton, Canada

Research Institute Liaison Chairs
Juho Perälä, VTT Technical Research Centre of Finland, Finland
Alexander Klaus, Fraunhofer Institute for Experimental Software Engineering (IESE), Germany
Kazumi Hatayama, Nara Institute of Science and Technology, Japan
Alin Stefanescu, University of Pitesti, Romania
Vladimir Rubanov, Institute for System Programming / Russian Academy of Sciences (ISPRAS), Russia
Tanja Vos, Universidad Politécnica de Valencia, Spain

Industry Chairs
Abel Marrero, Daimler Center for Automotive IT Innovations - Berlin, Germany
Sebastian Wieczorek, SAP AG - Darmstadt, Germany
Eric Verhulst, Altreonic, Belgium

VALID 2011

Committee

Dragos Truscan, Åbo Akademi University - Turku, Finland
Eric Verhulst, Altreonic, Belgium
Bart Vermeulen, NXP Semiconductors, The Netherlands
Arnaud Virazel, Université de Montpellier 2 / LIRMM, France
Tanja Vos, Universidad Politécnica de Valencia, Spain
Stefan Wagner, Technische Universität München, Germany
Melanie Ware, University of Ulster - Jordanstown, UK
Sebastian Wieczorek, SAP AG - Darmstadt, Germany
Cemal Yilmaz, Sabanci University - Istanbul, Turkey
Zeljko Zilic, McGill University, Canada

Table of Contents

Using Assertion-Based Testing in String Search Algorithms 1
Ali Alakeel and Mahmoud Mhashi

An Approach to Modularization in Model-Based Testing 6
Teemu Kanstren, Olli-Pekka Puolitaival, and Juho Perala

Dealing with Challenges of Automating Test Execution 14
Vitalina Turlo and Valery Safronau

Detecting Equivalent Mutants by Means of Constraint Systems 21
Simona Nica, Mihai Nica, and Franz Wotawa

Answer-Set Programming as a new Approach to Event-Sequence Testing 25
Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jorg Puhrer, Hans Tompits, and Cemal Yilmaz

A Test Case Suite Generation Framework of Scenario Testing 35
Ting Li, Zhenyu Liu, and Xu Jiang

Is Mutation Testing Scalable for Real-World Software Projects? 40
Simona Nica, Rudolf Ramler, and Franz Wotawa

Testing As A Service for Component-based Developments 46
Hien Le

A Zone-based Reachability Analysis of Variable Driven Timed Automata 51
Omer Nguena-Timo and Antoine Rollet

Retrospective Project Analysis Using the Expectation-Maximization Clustering Algorithm 58
Steffen Herbold, Jens Grabowski, and Stephan Waack

Extracting and Verifying Viewpoints Models in Multitask Applications 64
Selma Azaiez, Belgacem Benhedia, and Vincent David

Requirements and Solutions for Tool Integration in Software Test Automation 71
Bernhard Peischl, Rudolf Ramler, Thomas Ziebermayr, Stefan Mohacsi, and Christoph Preschern

RobusTest: Towards a Framework for Automated Testing of Robustness in Software 78
Ali Shahrokni and Robert Feldt

Simulated Injection of Radiation-Induced Logic Faults in FPGAs 84

Cinzia Bernardeschi, Luca Cassano, Andrea Domenici, Giancarlo Gennaro, and Mario Pasquariello

Concurrent Engineering used to Implement Risk & Hazard Control 90
Gheorghe Florea and Luiza Ocheana

Model Reconstruction: Mining Test Cases 97
Edith Werner and Jens Grabowski

Generic Data Format Approach for Generation of Security Test Data 103
Christian Schanes, Florian Fankhauser, Stefan Taber, and Thomas Grechenig

A Classification for Model-Based Security Testing 109
Michael Felderer, Berthold Agreiter, Philipp Zech, and Ruth Breu

Utilizing Domain-Specific Modelling for Software Testing 115
Olli-Pekka Puolitaival, Teemu Kanstren, Veli-Matti Rytky, and Asmo Saarela

Comparison of off-chip interconnect validation to field failures 121
Michael Shepherd, David Blankenbeckler, and Adam Norman

Software Testing in Critical Embedded Systems: a Systematic Review of Adherence to the DO-178B Standard 126
Jacson Rodrigues Barbosa, Marcio Eduardo Delamaro, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi

Using Assertion-Based Testing in String Search Algorithms

Ali M. Alakeel[1] and Mahmoud M. Mhashi[2]
College of Computers and Information Technology
University of Tabuk
P.O.Box 1458, Tabuk 71431, Saudi Arabia
alakeel@ut.edu.sa[1]
mmhashi@ut.edu.sa[2]

***Abstract*—Software programs may contain faults that cause them to work improperly. Assertion-Based testing has been shown to be effective in detecting program faults as compared to traditional black-box and white-box software testing methods. String search algorithm problem is one of the most important problems that had been investigated by many studies to find all the occurrences of a pattern (with size m characters) occurs in text (with size n characters), where m<<n. String search algorithms are one of the main elements of Information Retrieval Systems which are found in a wide range of applications such as military applications, aircraft software, medical applications, and commercial applications. Therefore, the correctness of any string search algorithms is vital. Different errors might occur during the implementation of any of these algorithms. An example of error, if the shift distance becomes zero, then the algorithm will not move forward. In this research, we show that Assertion-Based software testing may be effective in uncovering software faults associated with string searching algorithms. Our preliminary experimentation with this approach shows that several types of errors associated with string searching algorithms are uncovered using Assertion-Based software testing.**

Keywords-Software testing; Assertion-Based Testing; Program Assertions; String Search Algorithms.

I. Introduction

Software programs may contain faults that cause them to work improperly. The effects of software failure could be very disastrous and life threatening. For example, a software failure may cause an airplane to crash, a nuclear factory to meltdown or even to cause a military missile to hit the wrong target, e.g., [22]. For this reason, software testing methods has gained so much attention from researchers and industry practitioners since computers were invented.

Software testing is a very labor intensive task and cannot by any means guarantees the correctness of any software or that the software is error-free. However, thorough and rigorous software testing may increase the confidence in the software under test. There are two main approaches to software testing: Black-box and White-box. Test data generation is the process of finding program input data that satisfies a given criteria. Test generators that support black-box testing create test cases by using a set of rules and procedures; the most popular methods include equivalence class partitioning, boundary value analysis, cause-effect graphing. White-box testing is supported by coverage analyzers that assess the coverage of test cases with respect to executed statements, branches, paths, etc.

Programmers usually start by testing their software using black-box methods against a given specification. By their nature black-box testing methods might not lead to the execution of all parts of the code. Therefore, this method may not uncover all faults in the program. To increase the possibility of uncovering program faults, white-box testing is then used to ensure that an acceptable coverage has been reached, e.g., branch coverage.

Assertion-Based testing [4-5, 7] has been shown to be effective in detecting program faults as compared to traditional black-box and white-box software testing methods. Given an assertion *A*, the goal of Assertion-Based testing is to identify program input for which *A* will be violated. The main aim of Assertion-Based Testing is to increase the developer confidence in the software under test. Assertion-Based Testing is intended to be used as an extra and complimentary step *after* all traditional testing methods have been performed to the software. Assertion-Based Testing gives the tester the chance to think deeply about the software under test and to locate positions in the software that are very important with regard to the functionality of the software. After locating those important locations, assertions are added to guard against possible errors with regard to the functionality performed in these locations.

String search algorithms are one of the main elements of Information Retrieval Systems (IRS) which are found in a wide range of applications such as military applications, aircraft software, medical applications, and commercial applications. If an information retrieval system fails to return the correct piece of information, the results could be disastrous. For example, if a medical information retrieval system fails to return the *exact* prescribed medicine, this action may jeopardize the patient's life. Also, if a missile control system fails to retrieve the exact coordinates of the target, the results could be disastrous. Therefore, the correctness of any string search algorithms is vital. String searching algorithms are so fundamental that most computer programs use them in one form or another.

In this paper, we show that Assertion-Based software testing [4-5, 7] may be effective in uncovering software faults associated with string searching algorithms. Since the time Assertion-Based testing was proposed in [4] and to the best of our knowledge, this research is the first to present an application a proposed testing methodology, i.e., Assertion-Based testing, to a well known reported algorithms, i.e., string search algorithms. For the purpose of this research we have selected a number of well-known published string searching algorithms, gave them to programmers to implement them and then used Assertion-Based software testing to test these implementations. Our result is presented in a case study presented in Section IV of this text. It should be noted that the efficiency, performance or the competency of each string search algorithm, considered in our study, are not questioned. Our main objective is to show that Assertion-Based testing may be effective during the development and testing of such algorithms.

The rest of this paper is organized as follows. Related work is discussed in Section II. Section III presents an application of Assertion-Based software testing method to the string search algorithms. In Section IV, we present a case study, followed by our conclusion and future work in Section V.

II. Related Work

A. *String Search Algorithms*

Exact string searching is one of the most important problems that had been investigated by many studies, e.g., [8-21]. The problem of string searching may be stated as follows. Given a text string (Text) of size n and a pattern string (Pat) of size m (where n >> m), find all occurrences of *Pat* in *Text* [8].

As reported in the literature, many exact string searching and pattern matching algorithms were introduced and their performance was investigated against classical exact string searching algorithm such as Naïve (brute force) algorithm and Boyer-Moore-Horsepool (BMH) algorithm. Some of these algorithms preprocess both the text and the pattern, e.g., [9] while others need only to preprocess the pattern, e.g., [10, 11]. In all cases, the exact string searching problem consists of two major steps: checking and skipping. The checking step itself consists of two phases:

1) A search along the text for a reasonable candidate string

2) A detailed comparison of the candidate against the pattern to verify the potential match.

Some characters of the candidate string must be selected carefully in order to avoid the problem of repeated examination of each character of text when patterns are partially matched. Intuitively, the fewer the number of character comparisons in the checking step the better the algorithm is. Different string search algorithms may differ in the way they implement the checking process, e.g., [12, 13]. After the checking step, the skipping step shifts the pattern to the right to determine the next position in the text where the substring text can possibly match with the pattern. The reference character is a character in the text chosen as the basis for the shift according to the shift table. Some string search algorithms may use one or two reference characters and the references might be static or dynamic [14, 15]. Additionally, some algorithms focus on the performance of the checking operation while others focus on the performance of the skipping operation [16]. The shift distance used may differ from one string search algorithm to another; it ranges from only one position in the Naïve algorithm, up to m positions in Boyer-Moore-Horspool algorithm [11], m+1 positions in Raita's algorithm [10], and up to 3m+1 positions in CSA algorithm [17].

From the previous discussion, it can be noticed that there are different factors and elements of string search algorithms that may lead to program errors during the implementations of these algorithms into real program's code. Some of these elements are the starting point of checking, the direction of checking, the skipping strategy, the number of static or dynamic reference characters, and different shift distances. Thus it is possible that errors might occur during the implementation of any string searching algorithm. For instance, the shift distance might become zero or the number of occurrences of *Pat* in *Text* found by the algorithm might be less than or greater than the actual occurrences of *Pat* in *Text*. In a case study, presented in Section IV, we found out that Assertion-Based testing may be effective in catching some of the faults associated with string search algorithms.

B. *Assertion-Based Softwre Testing*

Assertions are recognized as a supporting aid in revealing faults during software testing, debugging and maintenance, e.g., [1-7]. Therefore, many programmers place them within their code in positions considered error prone or have the potential to lead to software crash or failure. An Assertion specifies a constraint that applies to some state of computation. The state of an assertion is represented by two possible values: *true* or *false*. For example, *assert*(0<index<=100), is an assertion that constraints the values of some variable "index" to be in the range of 1 and 100 inclusive. As long as the values of "index" is within the allowed range the state of this assertion is *true*. Any other values beyond this range, however, will cause the state of this assertion to become *false* which indicates the violation of this assertion.

Many programming languages support assertions by default, e.g., Java and Perl. For languages without built-in support, assertions can be added in the form of annotated statements. For example, [4] presents assertions as commented statements that are pre-processed and converted into Pascal code before compilation. Many types of assertions can easily be generated automatically such as boundary checks, division by zero, null pointers, variable overflow/underflow, etc. Beyond simple assertions that can easily be generated automatically, reference [4] presents a method to generate more complex assertions for Pascal programs. For this reason and to enhance their confidence in their software, programmers may be encouraged to write more programs with assertions.

It should be noted that writing the proper type of assertions and choosing the proper locations to inject them into programs depend heavily on the tester's experience and knowledge of the program under test. As mentioned previously, a simple tool may be used to automatically generate assertions in certain locations of the program which guard against errors such as division by zero, array boundary violations, uninitialized variables, stack overflow, null pointer assignment, pointer out of range, out of memory (heap overflow), and integer / float underflow and overflow [3]. However, there are application-specific locations in the program itself that may need to be guarded by assertions depending on the importance of these locations to the correctness of the application. For example, in string searching algorithms, computing the location of the pattern in the input string and index manipulation during the checking and skipping process are very important to the correctness of such algorithms.

```
#include <iostream>
#include <iomanip>
#include <cstring>
using std::cout;
using std::cin;
#define ASIZE 300
#define XSIZE 300
void preBmBc(char *x, int m, int bmBc[]) {
 int i;
for (i = 0; i < ASIZE; ++i)
bmBc[i] = m;
for (i = 0; i < m - 1; ++i){
/* A1: (x[i]>=0 and x[i]<ASIZE) */                    // Assertion No. 1
bmBc[x[i]] = m - i - 1;
}
}
void suffixes(char *x, int m, int *suff) {
int f, g, i;
suff[m - 1] = m;
g = m - 1;
for (i = m - 2; i >= 0; --i) {
/* A2: (i + m - 1 - f)>=0 and (i + m - 1 - f)<XSIZE) */ // Assertion No. 2
 if (i > g && suff[i + m - 1 - f] < i - g){
/* A3: (i)>=0 and (i)<XSIZE) */                        // Assertion No. 3
            suff[i] = suff[i + m - 1 - f];
  }
else {
if (i < g)
g = i;
f = i;
while (g >= 0 && x[g] == x[g + m - 1 - f])
      --g;
/* A4: (i)>=0 and (i)<XSIZE) */                       // Assertion No. 4
suff[i] = f - g;
   }
  }
}
void preBmGs(char *x, int m, int bmGs[]) {
int i, j, suff[XSIZE];
suffixes(x, m, suff);
for (i = 0; i < m; ++i)
bmGs[i] = m;
j = 0;
for (i = m - 1; i >= 0; --i)
if (suff[i] == i + 1)
for (; j < m - 1 - i; ++j)
if (bmGs[j] == m)
bmGs[j] = m - 1 - i;
for (i = 0; i <= m - 2; ++i){
/* A5: (m - 1 - suff[i])>=0 and (i)<XSIZE) */         // Assertion No. 5
bmGs[m - 1 - suff[i]] = m - 1 - i;
}
}
void BM(char *x, int m, char *y, int n) {
int i, j, bmGs[XSIZE], bmBc[ASIZE];
  /* Preprocessing */
preBmGs(x, m, bmGs);
preBmBc(x, m, bmBc);
  /* Searching */
j = 0;
while (j <= n - m) {
for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
if (i < 0) {
cout<<"\nAn occurrence at location  "<<j;
j += bmGs[0];
   }
else{
 /* A6: (i>=0 and i<XSIZE) */                         // Assertion No. 6
/* A7: ((y[i + j])>=0 and (y[i + j])<ASIZE) */        // Assertion No. 7
if(bmGs[i] > bmBc[y[i + j]] - m + 1 + i)
            j += bmGs[i];
            else
            j += bmBc[y[i + j]] - m + 1 + i;
  }
}
}
int main()
{
char Text[] = "test This is a test for string test";
char Pat[] = "test";
int m = 4;
int n = 35;
cout << "\nInput text: " << Text << "\nPattern: " << Pat;
BM(Pat, m, Text, n);
cout<< "\n Press ENTER to exit ....";
getchar();
return 0;
}
```

Figure 1. Boyer-Moore Algorithm with Assertions

During our case study presented in Section IV, assertions were injected in locations that are error-prone and crucial to the correctness of a string search algorithm. This knowledge is gained through our investigation of each string search algorithm considered during our experiment.

III. Using Assertion-Based Testing in String Search Algorithms

In this section, we present an application of Assertion-Based software testing to the Boyer–Moore string search algorithm [11]. Our complete case study is presented in section IV.

A. *Boyer–Moore Algorithm*

The Boyer–Moore string search algorithm [11] is a particularly efficient string searching algorithm. It has been the standard benchmark for the practical string search literature. The algorithm preprocesses the target string (key) that is being searched *for*, but not the string being searched *in* (unlike some algorithms that preprocess the string to be searched and can then amortize the expense of the preprocessing by searching repeatedly). The execution time of the Boyer-Moore algorithm, while still linear in the size of the string being searched, can have a significantly lower constant factor than many other search algorithms: it doesn't need to check every character of the string to be searched, but rather skips over some of them. Generally the algorithm gets faster as the key being searched for becomes longer. Its efficiency derives from the fact that with each unsuccessful attempt to find a match between the search string and the text being searched, it uses the information gained from that attempt to rule out as many positions of the text as possible where the string cannot match. Figure 1 shows an implementation of Boyer-Moore Algorithm after assertions have been added to it. In this program we inserted a total of seven assertions in different positions of the code. Assertions are numbered and shown in bold in Figure 1.

After applying Assertion-Based Testing to Boyer-Moore Algorithm of Figure 1 Assertion #2 was *violated*. As described in [4], during Assertion-Based Testing, each assertion found in the program under test is converted into a corresponding code during a pre-processing stage. For example, Assertion#2 of Figure 1 will be converted into the following code:

```
18.1      if (!((i + m - 1 - f)>=0))
18.1.1        cout << "\nAssertion 2a Violation!";
18.2      if (!((i + m - 1 - f)<XSIZE))
18.2.1        cout << "\nAssertion 2b Violation!";
```

Considering the above segment of code, according to Assertion-Based Testing presented in [4], Assertion#2 is violated if either of statements 18.1.1 or 18.2.1 is executed. During our experiment, Assertion #2 was violated through the execution of statement 18.1.1. The violation of Assertion #2 has detected a fault in this program which is caused by the use of an uninitialized variable "f" used in statement#19 in Figure 1. Note that uninitialized variables might cause very serious bugs in the program due to the nondeterministic values those variables might take during the course of different program executions. It should also be noted that many forms of uninitialized variable go undetected by C++ compiler.

IV. CASE STUDY

For the purpose of this case study, we have selected, from the literature, seven different string searching algorithms. These algorithms are implemented in C++ by three programmers with more than 5 years of experience in software development. C++ language was selected because it is widely used in the industry in our area and in America. The programs are executed and tested using traditional software testing methods. Specifically the following software testing methods were used: black-box testing as represented by boundary value analysis and equivalence class partitioning while white-box testing is represented by branch coverage. Using these traditional software testing methods, we were not able to uncover any faults in any of the seven programs. As stated in [4], Assertion-Based testing is intended to be used as an extra and complimentary step *after* all traditional testing methods have been performed to the software. Assertion-Based Testing gives the tester the chance to think deeply about the software under test and to locate positions in the software that are very important with regard to the functionality of the software. After locating those important locations, assertions are added to guard against possible errors with regard to the functionality performed in these locations. Therefore, and in order to uncover any faults, we injected assertions in certain locations of each of the selected string search algorithms used in this study and then applied Assertion-Based Testing as described in [4]. As reported in Table I, we were able to uncover program faults, in all of the seven programs, which were left uncovered by traditional software testing methods or by tests performed by the original authors of those string matching algorithms.

Information presented in Table I may be interpreted as follows. Column#1 and Columun#2 show the name of the string search algorithm and the number of assertions inserted in each one, respectively. Column#3 shows the number of assertions that were violated during Assertion-Based software testing. For example, row#3 of Table I shows that in an implementation of the Horespool algorithm [18], a total of three assertions were inserted in this program. Two out the three assertions (66.7%) were violated during Assertion-Based testing. In this case study, the number of assertions ranges from 3 to 18 assertions. The percentage of assertion violations ranges from 5.5% to 66.7% and the percentage of the tested algorithms that contains faults was 100%. It should be noted that the result of this experiment might be different for different programs with different types of assertions. The purpose of this case study is only to show that Assertion-Based Testing [4-5, 7] may be effective in detecting program faults when applied to the considered set of string search algorithm implementations used in this experiment. We emphasize that the quality and the merits of the string search algorithms themselves are not questioned and is beyond the scope of this research.

TABLE I. CASAE STUDY RESULTS

Algorithm's Name	#Assertions	#Violations
Boyer-Moore Algorithm	7	1
CSA Algorithm	13	1
Horespool Algorithm	3	2
KR Algorithm	4	1
AXAMAC Algorithm	9	1
COLUSSI Algorithm	18	1

V. CONCLUSION and FUTURE WORK

In this paper, we presented a new approach in which Assertion-Based testing is utilized to find software faults associated with string searching algorithms. We performed a case study in which a set of well-known string search algorithms are implemented and tested. During this case study, assertions were inserted in selected locations of each subject program to guard against possible errors. The result of this case study is encouraging and shows that Assertion-Based software testing was able to uncover faults in these programs that were overlooked by traditional software testing methods such as black-box and white- box testing. This result indicates that Assertion-Based testing may be very effective during the development and testing of string search algorithm. For our future research, we intend to extend our case study to include a wider range of string search algorithms especially those which function as a part of bigger commercial applications.

REFERENCES

[1] Stucki L. and Foshee G., "New Assertion Concepts for Self-Metric Software Validation," Proceedings of the International Conference on Reliable Software, pp. 59-71,1975

[2] Yau S. and Cheung R., "Design of Self-Checking Software," Proceedings of the International Conference on Reliable Software, pp. 450-457, 1975.

[3] Rosenblum, D., "Toward A Method of Programming WithAssertions," Proceedings of the International Conference on Software Engineering, pp. 92-104, 1992.

[4] Korel B. and Al-Yami A., "Assertion-Oriented Automated Test Data Generation," Proc. 18th Intern. Conference on Software Eng., Berlin, Germany, pp. 71-80, 1996.

[5] Alakeel A., "An Algorithm for Efficient Assertions-Based test Data Generation," Journal of Software, vol. 5, No. 6, pp. 644-653, 2010.

[6] Korel B. and Alyami A., “Automated regression test generation,” Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, pp. 143 – 152, 1998.

[7] Alakeel A., “A Framework for Concurrent Assertion-Based Automated Test Data Generation,” European Journal of Scientific Research, Vol. 46, No. 3, pp. 352-362, 2010.

[8] Stephen G., "String Searching Algorithms", World Scientific, Singapore, 1994.

[9] Fenwick P., "Fast string matching for multiple searches", Software–Practice and Experience, Vol. 31, No. 9, pp. 815–833, 2001.

[10] Raita T., "Tuning the Boyer-Moore-Horspool String Searching Algorithm", Software Practice and Experience, Vol. 22, No. 10, pp. 879-844, 1992.

[11] Boyer RS. and Moore JS., "A fast string searching algorithm", Communications of the ACM, Vol. 20, No. 10, pp. 762–772, 1977.

[12] Ager M. S., Danvy O., and Rohde H. K., "Fast partial evaluation of pattern matching in strings", ACM/SIGPLAN Workshop Partial Evaluation and Semantic-Based Program Manipulation, San Diego, California, USA, pp. 3 – 9, 2003.

[13] Fredriksson and Grabowski S., “Practical and Optimal String Matching”, Proceedings of SPIRE'2005, Lecture Notes in Computer Science 3772, , pp. 374-385, Springer Verlag, 2005.

[14] Smith P., "On Tuning the Boyer-Moore-Horspool String Searching Algorithm", Short Communication, Software Practice and Experience, Vol. 24, No. 4, pp. 435-436, 1994.

[15] Mhashi M., "The Effectof Multiple Reference Characters on Detecting Matches in String Searching Algorithms," Software Practice and Experience, Vol. 35, No. 13, pp. 1299 -1315, 2005.

[16] Mhashi, M., "The Performance of the Character-Access On the Checking Phase in String Searching Algorithms", Transactions on Enformatica, Systems Sciences and Engineering, Vol. 9, pp. 38 –43, 2005.

[17] Mhashi M. and Alwakeel M, “New Enhanced Exact String Searching Algorithm” IJCSNS International Journal of Computer Science and Network Security, Vol. 10, No. 4, pp. 13 – 20, 2010.

[18] Horspool R.N., “Practical fast searching in strings,” Software - Practice & Experience, Vol. 10, No. 6, pp. 501-506, 1980.

[19] Karp R.M. and Rabin M.O., 1987, “Efficient randomized pattern-matching algorithms, “IBM J. Res. Dev., Vol. 31, No. 2, pp. 249-260, 1987.

[20] Apostolico A. and Crochemore M., “Optimal canonization of all substrings of a string,” Information and Computation, Vol. 95, No. 1, pp. 76-95, 1991.

[21] Colussi L., “Correctness and efficiency of the pattern matching algorithms,” Information and Computation, Vol. 95, No. 2, pp. 225-251, 1991.

[22] [http://www.pcworld.com/article/110035/software_bug_may_cause_missile_errors.html]. Last access date: July 26, 2011.

An Approach to Modularization in Model-Based Testing

Teemu Kanstrén, Olli-Pekka Puolitaival, Juho Perälä
VTT Technical Research Centre of Finland
Oulu, Finland
{teemu.kanstren, olli-pekka.puolitaival, juho.perala}@vtt.fi

Abstract—**Test models in model-based testing are typically represented as state machines in terms of states and transitions. These states and transitions also are typically the focus of the test modeling approaches. Yet these test models are basically software components for the test automation domain, and need to be considered from test automation and software engineering viewpoints. In this paper, we describe a modeling approach that takes better into account these viewpoints. Taking these viewpoints into account, we propose a modularization approach for modeling in model-based testing and present a tool for supporting this modularization approach.**

Keywords-model based testing; test automation; modularization

I. Introduction

Model-based testing (MBT) is an advanced test automation technique focused on generating test cases from state-based models. In recent years several MBT tools have been presented and the industrial adoption of MBT techniques has been increasing [1,2]. The underlying modeling approach in these different tools is typically state-based, augmented by some programming language constructs to embed test instructions inside the test model to produce executable test cases from the state-machine transitions. The test generation is guided by the algorithms analyzing the model and parsing these programming language constructs to form test sequences and test data.

As such, this modeling approach can be seen as similar to other programming tasks. State transitions are executed to move from one state to another and these executions are performed in terms of programming language instructions embedded in these transitions. Yet the test modeling approach is based almost solely on state-machine notions–transitions between states and guard statements defining when transitions are allowed. While it is recognized that different domains and abstraction levels are important in MBT (e.g., [3]), the domain of test automation in itself is not considered in the common MBT modeling approaches. Relations between the test models and other software engineering artefacts are sometimes considered (e.g., [4]) but not the composition of the test models themselves. As these test models are in practice software components in themselves, we believe it is possible to provide a more efficient test modeling approach by introducing good practices from the software engineering domain, and specifically the software test automation domain into the test modeling approach itself.

Based on this background, we present a modularization approach for test modeling in MBT. This includes further modularization of the different traditional state-machine elements (transitions and guards) as well adding new ones specific to test automation (test oracles as specific transitions). We further present a modeling approach for describing test input and expected output in terms of a taxonomy of runtime invariance as described by our earlier work [5]. Finally, we identify a set of additional topics for future study that we believe will enable taking these approaches further. Similar to our inspiring domain of generic software engineering, we believe the end result helps achieve easier test model creation, evolution and maintenance. This forms a basis for more effective test modeling. The approach is implemented in a tool called OSMOTester, available as open-source [6].

The rest of the paper is structured as follows. Section II describes the background concepts relevant to this paper. Section III describes our test model modularization approach. Section IV discusses the concept in a wider context. Finally, conclusions summarize the paper and future works.

II. Preliminaries

This section introduces the background concepts relevant for this paper.

A. Modularization

Modularization is one of the basic concepts in software engineering in general. Van der Hoek and Lopez [7] provide an overview of software modularization and its different aspects in software engineering. They show how modularization has been considered important from the early days of programming and has since evolved to all aspects of modern software engineering, and continues to be an important research question. The aspects they describe include programming languages, software architectures and software evolution. They also note the need to avoid excess modularization where not necessary. In terms of addressing modularity, van der Hoek and Lopez [7] list a number of benefits such as reduced complexity, enabling parallel work, enabling evolution (easier understanding, resilience to change, etc.), and easing reuse. The cost is described in terms of requiring added effort for composition of the modular pieces to form a whole.

While this shows that modularity is considered important and is addressed in many software engineering domains, the consideration in MBT has focused on state-machine concepts [8] and not considered modularization in terms of test automation concepts and model elements generally. These are the concepts we address in this paper, in providing extended means to express modularity in test models, including important test automation concepts, and in mitigating the costs by providing automated support for module composition.

Considering modularity in software engineering in general, Sarkar et al. [9] list seven main properties of modularity. *Similarity of purpose* refers to grouping together elements related to providing a specific service. *Encapsulation* refers to encapsulating the internals of a module from its environment and external collaborators. *Compilability* refers to possible issues in compiling a module due to issues such as circular dependencies. *Extensibility* refers to providing means to extend a specific module without accessing its internals. *Testability* refers to the ease of testing the module. *Cyclic dependencies* negate many of the benefits of modularization and as such need to be avoided. *Module size* should be overall roughly equal. In the following sections we present our approach in Section IV also discuss how it relates to these properties.

B. Example System

Throughout the rest of this paper, we will use a simple vending machine example to illustrate the concepts discussed. This vending machine is a modified version of the example used in [2]. The relevant part of the operation of the machine can be described as the set of following properties:

- Accepts 10, 20, and 50 cent coins.
- When a total of 100 cents have been inserted the action *vend* is enabled.
- When *vend* is enabled, no more coins can be inserted (this assumption is relaxed later).
- When *vend* is activated, a bottle is produced, reducing the total number available and resetting the number of inserted cents to 0.
- When the machine is empty (no bottles), all actions are disabled.

Notice that for the sake of providing a concise example, this model simplifies several aspects such as providing change to the user when going over the total of 100 cents.

C. Model-Based Testing

The term model-based testing (MBT) can be defined in different ways. We follow Utting and Legeard [2] who describe MBT as "Generation of test cases with oracles from a behavioural model". The model describes the expected behaviour of the system under test (SUT), and is used by a MBT tool in order to generate test cases, in a form suitable for the test target, such as method invocation sequences and input data. The SUT output is checked by the test oracles also encoded into the model.

A typical model applied in the context of MBT is based on some form of states and transitions (sometimes referred to as pre/post conditions, historical or functional notations [8]), such as an extended finite state machine (EFSM). This describes the system behaviour in terms of states and transitions between these states. Basically a state can be described as a relevant combination of system internal variables. A transition forms an invocation of some functionality of the SUT, possibly affecting the observed state. Finally, guard statements over the transitions impose constraints on when a transition is allowed to happen.

To illustrate these concepts, Figure 1 shows an example snippet of a state-machine for the vending machine. In this example, *state 1* is where the machine includes 10 bottles and 100 cents have been inserted. As defined by the example guard statements, the *vend* transition to *state 2* is allowed if there are bottles available in the machine and 100 cents have been inserted.

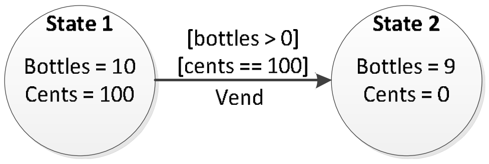


Figure 1. Vending machine model snippet.

In this case, both of these conditions have been met and thus this transition is enabled. Once it is taken, the number of bottles is reduced to 9 as one is deducted and the number of inserted cents is reset to zero. In this example, state is composed of the number of bottles available as well as the number of inserted cents.

To produce suitable test cases from such a model, this information is then transformed to test data for the SUT as defining the input to be given to invoke this transition on the SUT. This can be, for example, a message to the vending machine to produce a *vend* action. This is information encoded into the model by the user.

The test model must also define the expected output for each taken transition (the test oracle) in order to validate the correctness of the responses from the SUT for the given input. For example, the *vend* request should impact the reported number of bottles and produce a response as the "bottle" itself. This can be encoded in the test model along with the transition as an expected result, forming a test oracle.

Thus, to produce a suitable test model, the model must embed in itself means to identify test input to stimulate the SUT, the expected output for each input to produce a test oracle that gives verdict if the test passes or not, and a test harness that actually links the execution of the tests with the actual SUT.

III. TEST MODEL MODULARIZATION

The typical EFSM modeling notation for MBT was presented in the previous section. From this, we can list the following components that are needed to produce a suitable test model for test generation.

- Transitions to define what are the possible test steps for a specific SUT in its current state.
- A representation of the SUT state for the model.
- Test oracles to check the correctness of the received output and the internal state of the SUT when different transitions are taken.
- Guard statements to define when a specific transition is allowed to be taken.
- Test input to be linked to each transition/test step.

A. Traditional Modeling Approach for MBT

Figure 2 shows an example snippet of a test model for the vending machine, using the notation of the OSMOTester

MBT tool. This is based on existing works such as ModelJUnit described in [2], modified to better address the modularization aspects discussed in this paper. In this example, the variable *sut* represents the system and can either directly delegate the commands to the SUT itself or to a scripter writing a test script for later execution. The internal state of the model is composed of the *cents* and *bottles* variables. The guards are methods identified by the *@Guard* annotation and providing a Boolean value (*true* for allowing the transition). A transition is matched to its guard by the name given for both the *@Transition* and *@Guard* annotations.

```
private int cents = 0;
private int bottles = 10;
private VendingMachine sut = new VendingMachine();

  @Guard("10cents")
  public boolean allow10cents() {
    return cents <= 90 && bottles > 0;
  }

  @Transition("10cents")
  public void insert10cents() {
    sut.insert(10);
    cents += 10;
    assertEquals(cents, sut.cents());
  }

  @Guard("vend")
  public boolean allowVend() {
    return cents == 100;
  }

  @Transition("vend")
  public void vend() {
    sut.vend();
    cents = 0;
    bottles--;
    assertEquals(cents, sut.cents());
    assertEquals(bottles, sut.bottles());
  }
```

Figure 2. Example model snippet.

Test oracles are represented inside the transitions by the *assertEquals()* method calls (here from the JUnit test framework [10]) that compare the state of the test model to the state of the SUT. Input is given to the SUT in each transition. For example, *sut.insert(10)* in the “10cents” transitions, where 10 represents the number of cents inserted.

This is an example of the traditional approach to MBT, where every transition encodes all test components, including test input, and test oracles. In this approach, there is also a direct mapping from a single guard to a single transition. This is how traditionally most MBT tools expect the test models to be provided and how they process them (see e.g., [1] for comparison).

Figure 2 also illustrated two basic aspects of a test model in the terminology of this paper. What we term as control-flow in this aspect is the way the MBT tool traverses the EFSM expressed by this model, in evaluating the guards and taking suitable transitions as chosen by the active test generation algorithm. Together with this, we use the term data-flow to describe how the state variable values of the model evolve as the MBT tool traverses over the control-flow. For the vending machine, this translates to the evolution of the *cents* and *bottles* variables over time.

B. Modularizing the Control-Flow Modeling

Besides representing the guards, transitions and states of the EFSM as their own components, the traditional approach presented in the previous subsection is not very modular. It does not consider the separation of the different aspects of test inputs and test oracles (and associated test output). Furthermore, by assuming a direct one-to-one mapping from guards to transitions, the flexibility of the EFSM modeling is limited. As these test models are in practice software components themselves, this leads to several problems from their evolution and maintenance viewpoints such as duplication, low cohesion and weak separation of concerns.

To address these issues, we introduce new test model components and refine existing components. This includes more advanced guard composition, extensions for more explicit test oracles as components of the test model itself, and objects for generating input data and evaluating output data. In this subsection we discuss the control-flow elements and in the following subsection the data-flow elements.

The typical approach to modeling guard statements in an EFSM is to provide a specific guard attached to a specific transition as illustrated by Figure 2, where both the *10 cents* and *vend* are transitions and have a single dedicated matching guard statement. Taking the guard statement for *10 cents* as an example, it provides assertions over two separate concepts, the number of bottles and the number of cents. To help separate these concerns and provide a manageable modeling notation, we extend guard modeling by allowing decomposition of guard statements for a transition into several separate guards, and to share a single guard statement across several transitions as needed.

The decomposition aspect is illustrated in Figure 3. This decomposition provides for more cohesive structure where different concerns are addressed by different guards. In this case, the checks over the different state variables have been split into separate guard statements, each mapping to their specific transition by their name (e.g., “10cents”). The end result is more cohesive guard and better separation of model concerns.

```
@Guard("10cents")
 public boolean checkMaxCents() {
   return cents <= 90;
 }

 @Guard("10cents")
 public boolean checkBottles() {
   return bottles > 0;
 }
```

Figure 3. Guard decomposition.

However, decomposition alone does not fully address the need for providing cohesive guard statements over all the different transitions in the model. For example, if we consider the vending machine example, there are several transitions for inserting different types of coins (*10, 20, 50 cents*). In practice, none of these or the *vend* transition, should be allowed to execute if there are no bottles available. To support this, we need to provide specific shared guard statements.

This is illustrated in Figure 4 where the first guard *checkBottlesExist()* is shared by all transitions in the model, and the second one is shared by the three listed transitions (*10, 20, 50 cents*). The first of these examples is an example suitable for our simple vending machine example as is. The second one is an example of how we might model a common guard in a case where the user is allowed to insert enough coins for several bottles at once and we need to check that the total of inserted coins does not go over the number of available bottles.

```
@Guard
public boolean checkBottlesExist() {
  return bottles > 0;
}

@Guard({"10cents", "20cents", "50cents"})
public boolean allowMoreCoins() {
  return bottles >= (cents/100);
}
```

Figure 4. Shared guard example.

In addition to having guards and transitions govern how test sequences are generated, we have to consider the evaluation of the test results. This is done by a test oracle and is traditionally part of each transition as shown in Figure 2 (the assert statements). In many cases, specific checks are needed for transitions to check their specific results. However, it is also commonly important to evaluate the general state of the model against the matching state in the SUT. To support more explicit modeling of test oracles, we extend our modeling notation to add general test oracles for program state over several transitions. These are similar in decomposition to the shared guards and identified by the *@Oracle* annotation. Figure 5 illustrates this with an example for the two state variables shown in Figure 2 for the vending machine. In practice, our MBT tool sees these as specific transitions to be executed between other transitions. It is also possible to relate them to specific transitions with the style of *@Oracle("transition-name")* similar to guards. In our models, generic oracles apply regardless of existence of specific ones. Any oracle matching a transition is always evaluated.

```
@Oracle
public void evaluateBottles() {
  assertEquals(bottles, sut.bottles());
}

@Oracle
public void evaluateCents() {
  assertEquals(cents, sut.cents());
}
```

Figure 5. Generic oracle example.

As these generic checks can be expressed separately and evaluated specifically by OSMOTester, they not only allow for the modular expression of generic test oracles but also add more power to the verification functionality of the test model and the MBT approach itself. In Figure 5, the state of the model and the state of the SUT are now evaluated to match continuously without the need to express them explicitly over each transition. Any deviation is thus captured as soon as it occurs and not possibly several transitions later in the *vend* transition (if at all) as was the case in Figure 2.

Finally, we also need to consider how and where test generation should be stopped. The typical approach in MBT is to describe the test model as a state machine with the expectation that test cases can be generated and the model can be traversed at different points, where test generation should practically always be enabled. The choice of what transitions to take and when to stop the test generation is mainly up to the test generation algorithm. However, there are points where it is possible that no transition is enabled and the typical modeling approach gives no indication to test generation as to how this should be evaluated. The generic algorithms used to generate tests from the test model cannot know how to evaluate this condition for a specific SUT and its test model. For example, in the case of the vending machine example, if the *vend()* transition is taken 10 times, the number of bottles will reach zero and the shared guard *checkBottlesExist()* will cause a state where no new transitions are available. At this point, the test generation tool cannot know if this should be treated as a failure or as a clue to end test generation for this step.

To enable the model to express this kind of information, we add a new annotation called *@EndCondition* and as illustrated in Figure 6. When a method with this annotation returns *true,* it is taken as an indicator that the current test generation from this model should be stopped and a new test case should be started. If no transitions are available and there is no *@EndCondition* that returns *true*, the current test case is reported by the tool as a failure. This will most likely indicate a problem in the test model itself. It should be noted that this annotation is not required for the test generation algorithms to stop test generation but they can also stop in other phases where found appropriate by the algorithm. It can also be used at any point to describe conditions to stop test generation, regardless of the state of the model.

We also provide specific notations for setting up new test cases and shutting down a running system using notations such as *@BeforeTest*, *@AfterTest*, *@BeforeSuite*, and *@AfterSuite*. We borrow these concepts from familiar tools such as JUnit [10] and TestNG [11], helping also to provide familiar concepts for other tool users. They basically define methods that are to be executed before and after a test case or a test suite respectively, regardless of the algorithms and end conditions.

```
@EndCondition
public boolean endIfNoBottles() {
  return bottles == 0;
}
```

Figure 6. Expressing end conditions.

Using our new control-flow modeling notations we can thus produce the model shown in Figure 7. Notice that there is now only a single test oracle where all test assertions are centralized. The transitions can now focus on performing actions on the SUT and updating the model state accordingly. Also, the transition *10cents* no longer requires a guard statement as it is fully covered by the shared guard statement (also applying to *vend*). Finally, the model can no longer enter an unknown state as the end condition for a state with no bottles is explicitly specified.

```
private int cents = 0;
private int bottles = 10;
private VendingMachine sut = new VendingMachine();

  @Guard
  public boolean checkBottlesExist() {
    return bottles > 0;
  }

  @Transition("10cents")
  public void insert10cents() {
    sut.insert(10);
    cents += 10;
  }

  @Guard("vend")
  public boolean allowVend() {
    return cents == 100;
  }

  @Transition("vend")
  public void vend() {
    sut.vend();
    cents = 0;
    bottles--;
  }

  @Oracle
  public void evaluateState() {
    assertEquals(bottles, sut.bottles());
    assertEquals(cents, sut.cents());
    assertTrue(cents >= 0);
    assertTrue(cents <= 100);
    assertTrue(bottles >= 0);
  }

  @EndCondition
  public boolean endIfNoBottles() {
    return bottles == 0;
  }
```

Figure 7. The model snippet in updated notation.

C. *Modularizing the Data-Flow Modeling*

The modeling notation described so far in the previous section shows how we can modularize the control-flow aspects of test modeling in MBT. From the viewpoint of data-flow we need to consider also the input- and output-data values and their respective constraints. Input data needs to be generated for the different parameters given to the SUT, and needs to respect the set of expected constraints for the SUT functions they are linked to. But since full coverage of most input combinations is not possible to achieve, we must also define a set of constraints to define what type of test data should be generated. The output must similarly consider the constraints for the output values received from the SUT as response to the provided stimuli (input).

To support modeling these data-flow constraints, we provide a generic library of objects we term as invariants objects. These are based on our previous work in identifying different aspects of runtime invariance in software behavior [5]. Each invariant object allows one to specify a set of constraints over the data value it represents and to use these as a basis to perform actions such as generate input data or evaluate the conformance of given (output) values. These allow for addressing data-flow invariance for a specific value in a single object, effectively modularizing the constraints over a single variable in a single object.

An updated model for the vending machine using this notation is shown in Figure 8. This time, the use of the invariant objects for data-flow representation allows for a compact representation, and this includes transitions for all possible coin types and vending. By adding the shared guard and end condition from Figure 7 the model will include all the important model elements for the vending machine. The invariant objects presented in the figure are specified for integer data types, and we currently support the basic data types of integers, floating points, Booleans and character strings. The constraints supported by these are defined according to the taxonomy presented in [5].

Note that this model slightly changes the expected behavior of the vending machine towards a more realistic one. This specification now accepts any number of coins and deducts 100 from the number of inserted coins when *vend* is applied. It also collapses all *insertXXCents()* transitions (*10, 20, 50*) from the previous models into a single one, where the input is represented by a single invariant object defining the allowed input values. This is the *ci* object (short for *centInput* for space reasons in the figure) for the input value definitions. Test oracle expectations for both *cents* and *bottles* variables are expressed by the *co* and *bo* variables (short for *centOracle* and *bottleOracle* for space reasons in the figure).

```
private IntInvariant ci = new IntInvariant();
private IntInvariant co = new IntInvariant();
private IntInvariant bo = new IntInvariant();

public void TestModel() {
  //set up allowed input values
  ci.addValue(10);
  ci.addValue(20);
  ci.addValue(50);

  //set up evaluation constraints
  co.setMin(0);
  bo.setMin(0);
  bo.setMax(10);
}

@Transition("insertCoins")
public void insertCents() {
  int coin = ci.input();
  sut.insert(coin);
  cents += coin;
}

@Guard("vend")
public boolean allowVend() {
  return cents >= 100;
}

@Transition("vend")
public void vend() {
  sut.vend();
  cents -= 100;
  bottles--;
}

@Oracle
public void evaluateState() {
  assertEquals(bottles, sut.bottles());
  assertEquals(cents, sut.cents());
  co.evaluate(cents);
  bo.evaluate(bottles);
}
```

Figure 8. Data-flow modularization.

D. *Further modularization support*

In the previous sections we have shown how to build a modularized test model in our notation. So far we have included the elements needed to build a useful model for generating test sequence and test data. Additionally, it is also important in test automation to be able to express how the generated test cases cover different requirements, a concept also supported by different MBT tools [2]. We support this through a special requirements object as illustrated in **Figure 9**. Note that defining the requirements twice is not required (add() and covered() methods) but doing so allows the tool to report the coverage percentage.

```
@RequirementsField
private Requirements req = new Requirements();
@TestSuiteField
private TestSuite s = null;

public TestModel() {
  req.add("vend");
}

@BeforeTest
public void startTest() {
  System.out.println("Starting Test "+s.count());
}

@Transition("vend")
public void vend() {
  req.covered("vend");
  …
}
```

Figure 9. Additional supported test model components.

Figure 9 also illustrates how the modeler can access the test generation history by adding a *TestSuite* field that will be initialized by the MBT tool before starting test generation. This provides useful information for evaluating and debugging the test model itself. It also allows the user access to the current test case object, for example, enabling the user to set the test case as passed or failed for any special reporting extensions in the MBT tool.

So far we have discussed several new notations for MBT that help produce more modular test models. However, an important question of decomposing the objects representing this notation remains. Besides expressing all elements in a single model, we also support decomposing the model objects into several sub-models. When these are registered into OSMOTester, it will parse them all and match all the expressed model elements into a single internal representation. This effectively allows one to, for example, represent the test oracles in one partial model, guards in another, transitions in a third, and the remaining ones in fourth. Merging is based on handling the model element naming across the different model objects as if they were one.

Finally, we also support the basic set of modular aspects for any MBT tool as discussed in [12]. It is possible to plug in different test generation algorithms, algorithms for defining length of generated test suites and test cases, and to attach various test harnesses and test analysis tools. Figure 10 shows an example of a test sequence visualization tool we provide as a plugin to the core OSMOTester itself, using as output a test listener interface provided by the core.

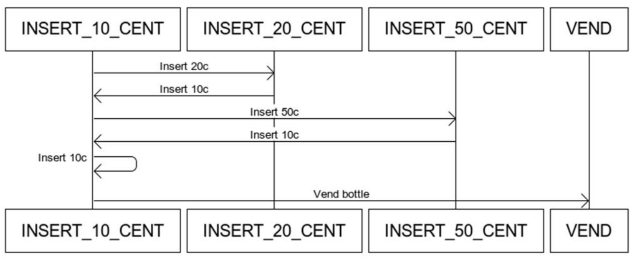


Figure 10. Test sequence visualization.

This visualization shows each transition in the test model as a box, and the sequences of transitions taken as arrows from one to another. In this case, the user has chosen to use "10cents" as the default state, which is why the arrows seem to originate from this box. This is just one of the available visualizations as an example of something that can be plugged in to describe the test cases.

E. *Modularization Summary*

Figure 11 illustrates the overall flow of the different control-flow modules in our model. Before test suite generation commences, *@BeforeSuite* annotated methods are executed. Before each new test generation, *@BeforeTest* annotated methods are executed. *@Guard* methods are checked for enabled transitions, of which one is picked by the test generation algorithm. After each transition any associated *@Oracle* methods are executed. If *@EndConditions* exist, they are evaluated for stopping criteria for a single test case. If not, the criterion is left to the test generation algorithm. *@AfterTest* methods are executed when a test generation stops, and *@AfterSuite* when all test generation stops.

Data-flow support is defined in terms of invariant objects defining constraints over data-flow values that support generating input and evaluating output. Coverage requirements can be expressed in the test model as objects of their own, and models can be built from separate model objects as best seen fit. We see further developments in terms of more complex combinations of data-flow and control-flow elements as discussed next.

In relation to the different properties of modularization that were discussed in section II.A, we improve on several of these properties. *Similarity of purpose* is supported by more explicit grouping of elements such as test oracles. *Extensibility* is supported by allowing composition of the model elements (test steps/transition components) from several different objects and linking them automatically together. It is possible to add new test oracles, guards and other elements as separate model objects without touching existing ones (also helping address similarity composition). We avoid *cyclic dependencies* and enhance *compilability* by keeping elements separate and associating by the transition name metadata only (vs. strict static linking). Our framework is *encapsulated* in a set of simple annotations, providing only minimal exposure on the test models. We make *module size* easier to manage with finer granularity of model elements. *Testability* is mostly handled in itself by generating tests from the model and executing them against the SUT as in MBT in general, verifying both the test model and the SUT.

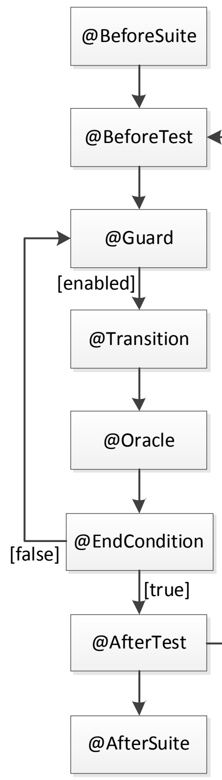


Figure 11. Control-Flow Summary.

IV. DISCUSSION

Modularization is one of the key aspects of good software engineering in general. As we have discussed and illustrated in the previous sections, test modeling in MBT is practically a software engineering activity in itself. It is basically about engineering a piece of software that generates a test suite in terms of the MBT framework, provided by available tools and libraries. While the traditional test modeling approaches for MBT have been lacking proper modularization mechanisms, we provide a set of means to achieve increased modularity in test modeling. This provides for increased separation of concerns, more cohesion and less duplication. As generally in software engineering, this helps achieve higher maintainability and supports model evolution.

While we advocate the use of our new modularized modeling notation, our approach also fully supports the more traditional modeling approach. It is possible to fully compose the model of transitions, guards and embedding all the information inside these without any modularization. This means just using a specific subset of our modeling notation (*@Transition* and *@Guard*). It is our experience that the best result in practice is to combine parts of the different approaches where most benefit can be gained. That is, modularizing the most common parts while keeping specific parts where it makes more sense according to the case at hand.

As our models are written in a standard programming language (Java), it is also possible to decompose the models into modules in terms of classes and methods. This can help achieve some of the benefits discussed here in itself (by using classes and objects), but it is also our experience that as the model is made more explicit in terms of generic test oracles, guards and invariant objects, this helps build a more explicit and understandable model. This also further helps in the model creation, evolution and maintenance, where human understanding is typically the key factor in software engineering. The use of a common programming language also helps more generic modularization as we can make use of the wide set of existing Java libraries.

Another aspect related to the modularization of test models is the modularization of the test models into representing different viewpoints of the SUT behavior. While the test models we have presented in this paper describe the expected correct behavior of the vending machine, another interesting aspect is the modeling of the failure behavior of the SUT. In case of the vending machine, this would include trying to insert incorrect values, access the vending functionality with less than 100 coins, using negative values, and so on. These different viewpoints can be modeled as separate models addressing these specific constraints along with matching test oracle definitions. This is a form of modularization itself.

Related to the *@Oracle* elements we have also found that it is useful to be able to access selected pre-transition state when checking the overall state after the transition. Similarly we have noted that this notation can be used for extended purposes such as supporting data collection for test reporting. Thus we are looking into options to extend this to support both with *@Pre* and *@Post* transition annotations as extensions of the current *@Oracle* approach.

In relation to the invariant objects, we described our support for test modeling in terms of data-flow invariants based on our earlier work on creating a taxonomy of runtime invariance in software behavior [5]. While we currently use this invariance to provide support for data-flow modeling, the taxonomy in itself is more extensive in describing also patterns over control-flow and various invariant scopes. This is a topic for future work in providing more extensive support for modeling software behavior.

Our experience thus far has been that we can modularize control-flow in terms of the generalized test oracles and guard conditions (in a way defining invariant control-flow patterns), but the more advanced support in terms of runtime invariance is challenging to express in a textual support in a way that is natural for human consumption. In terms of data-flow modeling, we also see the use of data-flow constraints to support test generation more widely in terms of boundary conditions, category partitions, and other relevant test data constraint and analysis definitions. This requires pairing data generation algorithms with the invariant constraint definitions. At the same time our experience has also been that using more domain friendly names (e.g., splitting integer invariants into value range and similar objects) is easier to understand and we are evolving the expressiveness of the data-flow elements into this direction. These are some of the more advanced research topics in relation to invariant objects that are out of the scope of the modularization approach described in this paper but relevant for future studies.

Similarly related to extending the modeling approach is the combination of different properties of invariance expressed in the model. For example, in the model presented in

Figure 8, the *ci* variable presents a set of input constraints for the generated test data related to the operations over the *coins* state variable. At the same time the *co* state variable presents a set of constraints over the expected values of the same state variable. Expressing these relations such as how the input should be constrained in relation to the current value of the state variable is challenging. This also applies to combining these invariants more generally over different control-flow aspects and data-flow aspects. Some viable approaches could be found in existing works such as combining evolutionary testing with MBT (e.g., [13]).

Many of these aspects come back to the limitations of the textual modeling approach in relation to the advanced concepts presented by the taxonomy in [5]. Thus one interesting aspect in relation to this is the application of visual modeling tools and domain-specific concepts. We have previously studied combining visual representations in terms of domain-specific models (DSM) to provide more visual support for test modeling [14]. Combining this to provide more intuitive support for representing complex interactions over the invariant properties is another interesting research topic for future studies. While DSM commonly considers the models it builds from the perspective of the application domain of a specific company, the modeling notation we describe here can also be seen as a form of a domain specific language for test modeling in the domain of MBT.

While we have so far discussed mainly aspects related to dynamic analysis and modeling related aspects, there are also several points where static analysis can be useful. This includes algorithms and techniques such as symbolic execution to generate test paths to reach defined requirements, test end conditions, automated input data boundary analysis, and other similar optimizations. These are aspects supported already in many advanced (commercial) MBT tools. So far, we have focused on the modularization of the dynamic runtime aspects. Extending this to the domain of static analysis of these models is out of the scope of our work but an interesting and relevant topic for future works.

We have applied our approach and tool on several commercial projects and keep evolving it according to our experiences in these projects. Due to the nature of the projects we cannot disclose their details. However, they have been successful in improving the aspects of model creation, management and evolution described here. Similarly, it has helped make the adoption of MBT approach easier, also leading to reduced costs in test automation.

V. Conclusions and Future Work

In this paper, we have presented a modularization approach for test models in model-based testing. This approach extends the traditional approach that focuses on the state-machine abstraction by considering common software engineering aspects and specific components of test automation. While the traditional approach focuses on state-machine abstractions in terms of guards and transitions, we extend this to include new state-machine elements for test models, including shared guards over several transitions, generic test oracles over the general system tests, and test end conditions. These are mainly related to the control-flow aspects of behaviour modeling, and we further provide added support for data-flow representations in terms of objects describing properties of runtime invariance over system behavior.

Finally, we identify potential topics for future works in terms of extending invariant object representations to control-flow aspects and their relation to the data-flow over different scopes (as identified in our previous work), and also more extensively in terms of test automation components such as input value boundary and category analysis. We also identify extensions of the modularization into the domain of static analysis, and providing more human-friendly modeling notations for complex models and invariant objects as interesting topics for future work.

VI. References

[1] O-P. Puolitaival, M. Luo, and T. Kanstrén, "On the Properties and Selection of Model-Based Testing Tool and Technique," in *1st Workshop on Model-Based Testing in Practice (MOTIP)*, 2008.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*.: Morgan Kaufmann, 2007.

[3] W. Prenninger and A. Pretschner, "Abstractions for Model-Based Testing," *Electronic Notes in Theoretical Computer Science*, vol. 116, pp. 59-71, 2005.

[4] Q. Farooq, M. Z. Zohaib, Z. Malik, and M. Riebisch, "A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support," in *17th IEEE In'tl. Conf. and Workshops on Engineering of Computer-Based Systems*, 2010, pp. 41-49.

[5] T. Kanstrén, "Towards a Taxonomy of Dynamic Invariants in Software Behaviour," in *2nd Int'l. Conf. on Pervasive Patterns and Applications (PATTERNS)*, 2010.

[6] T. Kanstrén. (2011, July) OSMOTester. [Online]. http://code.google.com/p/osmo/

[7] A. van der Hoek and N. Lopez, "A Design Perspective on Modularity," in *10th Int'l. Conf. on Aspect-Oriented Software Development*, Pernambuco, Brazil, 2011, pp. 265-279.

[8] M. Utting, A. Pretschner, and B. Legeard, "A Taxonomy of Model-Based Testing Approaches," *Software Testing, Verification and Reliability*, 2011.

[9] S. Sarkar, G. M. Rama, and A. C. Kak, "API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 14-32, Jan. 2007.

[10] (2011, July) junit.org. [Online]. www.junit.org

[11] (2011, July) testng.org. [Online]. www.testng.org

[12] V. V. Kuliamin, "Component Architecture of Model-Based Testing Environment," *Programming and Computer Software*, vol. 36, no. 5, pp. 289-305, 2010.

[13] F. Lindlar, A. Windisch, and J. Wegener, "Integrating Model-Based Testing with Evolutionary Functional Testing," in *Third Int'l. Conf. on Software Testing, Verification and Validation Workshops*, 2010, pp. 163-172.

[14] O-P. Puolitaival and T. Kanstrén, "Towards Flexible and Efficient Model-Based Testing, Utilizing Domain-Specific Modelling," in *10th Workshop on Domain Specific Modelling*, 2010.

Dealing with Challenges of Automating Test Execution

Architecture proposal for automated testing control system based on integration of testing tools

Valery Safronau and Vitalina Turlo
Software Testing Automation Department
Applied Systems Ltd.
Minsk, Belarus
safer@appsys.net, turlo@appsys.net

***Abstract*—If implemented correctly, automated software testing maybe an efficient way to circumvent time and resource shortages and ensure faster time to market for new products. Our experience and survey data show that the execution of automated tests is often accompanied by a number of time-consuming and routine operations that are performed manually, e.g., operating virtual machines, setup and cleanup of the environment, test launch, logging defects, etc. These menial chores can be automated with the help of simple command files or by developing an automated testing control solution. In the long run, the latter is a more efficient approach. The paper focuses on the challenges that companies face in attempting to build such a solution, and provides practical recommendations on implementation. Finally, we provide an architecture proposal for the system for automated testing based on testing tools integration, define its features and describe the interactions between its components.**

Keywords-Desktop application testing; survey; industrial experience; integration of testing tools; automated testing control solution.

I. Introduction

High quality, timely testing is crucial to the development of a reliable software product. By the same token, running regression tests on every released (stable) build is critical, especially in the case of continuously developed complex systems with extensive functionality. However, due to the shortage of resources, regression testing is often being neglected, and its significant lack or incompleteness is one of the greatest problems in software development quality assurance.

In order to solve this issue, Automated Testing (AT) is used. The fact is that implementing AT can be a great challenge in its own right, as it requires well-tuned software development and testing processes as well as clearly organized communication flows. “One of the primary reasons software testing tool implementations fail is because there is little or no testing process in place before the tools are purchased [1].”

In many instances the expression “automated testing” is misleading, as the testing process is still being controlled by a test engineer, especially where desktop applications are concerned. According to the online surveys conducted by Applied Systems Ltd. via the SurveyMonkey.com service, a tester has to manually fulfill some or all of the operations to execute automated tests on a new product build, such as configuring the testing environment, starting/shutting down Virtual Machines (VMs), launching tests, submitting bugs to a tracking system, closing fixed bugs, generating reports, and so on [2][3]. In addition to being very time-consuming, manual operations drastically increase the probability of human error. For these reasons, our goal is to enable the unmanned execution of the full AT cycle by completely automating these routine operations.

In this paper we describe a new, efficient approach to controlling automated software testing that meets the aforementioned challenges. The solution is based on the integration of testing tools. It has been applied in practice, and has proven useful in the automated testing of desktop applications, ensuring non-stop execution of tests while eliminating menial and boring tasks from the work of testers. One of the most obvious benefits of this solution is guaranteed regression testing of each new build.

The present work focuses on the realization of unmanned execution of automated tests – from environment set-up and test launch to defect tracking and report generation – but not on design and development of automated test scripts. We assume that test automation engineers know how to create tests that are reliable, maintainable and data-driven, while complying with the principles of test case independence, absence of redundant code, and scalability.

The findings of this paper are based on more than five years of practical experience in the automated testing of desktop software, as well as the results of two IT community surveys with a pool of more than 300 respondents.

Section 2 gives an overview of key previous work in the field of automated testing. In Section 3 we examine the evolution of automated testing and suggest a new classification of test automation levels emphasizing the amount of manual routine operations in the AT process. Section 4 is dedicated to exploring the main challenges inherent in building an Automated Testing Control System (ATCS). In Section 5 we propose the working archetype of such an ATCS with a detailed description of its main features and components. The Conclusion section summarizes the paper’s findings and outlines the field of research for future work.

The insights of the present work will be useful to Test Automation Engineers, Heads of Testing and QA departments, and those practitioners who wish to develop an in-house solution for automated testing control.

II. Related Work

As automated software testing gains popularity, the body of literature on the subject has been growing steadily in recent years. They provide test engineers with the theoretical and practical base necessary for a successful implementation of automated tests [1][4][5][6]. Authors with extensive professional experience in the industry guide the reader through the decision whether to automate tests, help to navigate through a plethora of testing tools to select the best fit ones, and give advice on building robust and documented testing processes [1]. The works also offer guidance on test planning, design, development, execution, and evaluation [4][6].

For a constructive discussion on which tests cases should be automated and guidelines for assessment of return on investment, see the work by Dustin and Garrett in [6] and [7], as well as Chapter 2 in Mosley and Posey in [1].

In *Automated Software Testing* Dustin, Rashka and Paul introduce the concept of Automated Test Life-Cycle Methodology (ATLM), "which is a structured methodology geared toward assuring successful implementation of automated testing [4]." They identify five phases of ATLM, namely:

1) Decision to automate test.
2) Automated test tool acquisition.
3) Introduction of automated testing to a new project and its optimization.
4) Test planning, design, development and execution.
5) Test evaluation.

Mosley and Posey argue that ATLM is an "artificial construct" that is not very useful for practitioners. They argue against the idea of a software testing life cycle, and claim that the result of the implementation of test automation depends on the quality of the processes already in place in the organization [1]. Despite certain differences in their approach to testing, Dustin and Mosley both promote a deliberate, well-reasoned preparation for test automation, including in-depth studies of test requirements, setting realistic expectations and planning for automated testing.

Our contribution to the existing knowledge on the topic consists in proposing an architecture design for automated testing control system, which is based on the integration of testing tools. We focus on how to realize completely unmanned test execution.

III. Evolution of Automated Software Testing

"Automated software testing" is a controversial expression employed by software companies regardless of the test automation level they have achieved.

Attempts to classify the levels of maturity of automated testing are not new. For instance, Dustin et al. correlate the four levels of automated testing described by Krause to the Software Testing Maturity Model (TMM) [4][8][9].

At the initial TMM level testing is not separated from debugging. It corresponds to "accidental automation," automated testing that is nonexistent or carried out on an ad-hoc or experimental basis. Test automation is not supported by process, planning and management activities; scripts are not reusable or maintainable.

At the second, Phase Definition level, testing and debugging are separated, and "incidental automation" occurs. At this phase automated scripts are adapted, but not reusable, and there are no defined processes.

The integration phase corresponds to a level of maturity where testing no longer follows coding, but is integrated into the software life cycle. At this stage, automated testing is referred to as "intentional automation." The process is well documented and well-managed; scripts' reusability and maintainability are at the core of test design and development.

At the fourth TMM level, testing is a measured and quantified process. Defects are tracked and assigned a severity level. In automated testing, this stage is called "advanced automation" and is supplemented with post-release defect tracking. The test team is an integral part of product development, which ensures that bugs are found as early as possible.

The classification of automated testing maturity levels that we suggest below does not conflict with the TMM model. However, we focus on a different criterion, which is the number of operations that are still being performed manually during automatic test execution. In addition, we emphasize such factors as organizational needs and project length and requirements.

In this section, we will define three stages of testing automation evolution as we view it and provide their principal characteristics (see Fig. 1).

A. *Infancy Stage*

This phase is marked by the emergence of scripts and automated tests. The scripts usually perform frequent, routine functions necessary to prepare the product for testing, e.g., the copying of product installation and configuration files to the testing PC and basic system setup. The scripts can also be used to verify particular product functionality. Along with the scripts, the automated tests created with special test automation tools (e.g., Visual Studio, HP QTP) appear in the testing process of an organization.

The main characteristics of this stage are:

- Lack of arranged test storage (generally, the tests are stored on the tester's PC and used solely by him or her, i.e., they are not reusable or adapted to any changes of tested interfaces).
- Need for systemized test launch.

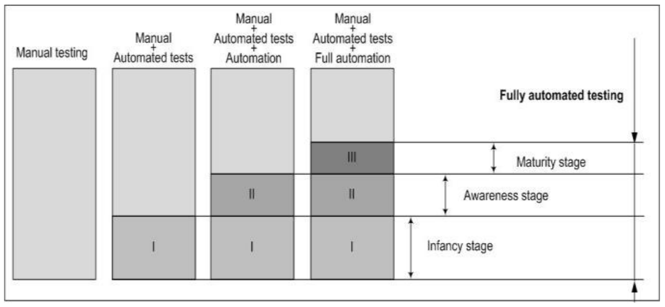


Figure 1. Stages of testing evolution.

- Shortage of documented procedures and common practices for interpreting test results and creating reports.

The infancy stage lacks a systematic approach to the integration of AT into the software quality assurance process.

At this point the tests are often unstable and their results cannot be reliable. However, they may free up a certain amount of resources by helping testing specialists fulfill the most routine tasks.

Typically, organizations that dwell at this phase have short-term projects, and thus lack the opportunity to upgrade to a higher level of test automation. These include small companies that have no testing process as such, as well as firms that are just beginning to use automated tests.

B. *Awareness Stage*

In this phase the majority of activities are automated using, for example, batch files: launching tests, starting and shutting down VMs, copying the necessary configuration files to the testing PC and other operations.

The following features are typical of the second stage:

- Improvement of test quality.
- Arrangement of centralized storage for tests and libraries of functions (tests become reusable).
- Tests are launched automatically upon the issue of each new product build.
- Naming rules for automated tests take effect.
- Guidelines for processing test results (submitting and closing bugs) are elaborated.

At this stage, which is the most widespread among companies, we may trace particular signs of automated testing. A typical company representing this phase is a developer of middle- and long-term software projects, which has well-established testing processes and realizes the need for regular regression testing.

C. *Maturity Stage*

This is the most advanced level of automated software testing, where it is seamlessly integrated into the company's testing processes. As Mark Fewster and Dorothy Graham put it, "A mature test automation regime will allow testing at the "touch of a button" with tests run overnight when machines would otherwise be idle [5]."

We characterize this stage as "full testing automation". By that we mean that **all** the operations related to test execution are done automatically, without the participation of a test engineer. These include:

- Starting and shutting down Virtual Machines (VMs) in cases using virtualization during testing.
- Configuring the testing environment.
- Queuing builds for testing according to their priority.
- Execution of tests upon successful build compilation.
- Submission of defects to the Bug Tracking System (BTS).
- Closing fixed bugs in the BTS (optional).
- Generation of a unified report on all passed tests.

Obviously, all of these elements should be automated to the extent that it is cost and time efficient [1]. Generally, such an advanced level of automation is attained by companies developing complex software products with extensive functionality. They are engaged in middle- and long-term projects and have to meet the challenges of missing or incomplete regression testing, and the effort of achieving the advanced level is worthwhile for them.

As we proceed, we will assume that introducing automated tests is a decided matter, its economical feasibility is proven, and a company's goal is to achieve the maturity stage where tests are executed automatically, i.e., without the interference of a test engineer. Many publications discuss the criteria according to which tests should be automated. They also provide techniques for evaluating return on investment of test automation [6][7]. These particular topics are beyond the scope of this paper.

This paper focuses on achieving the advanced level of automation by means of a special automated testing control solution. Below, we describe the main challenges of developing an ATCS and propose the software architecture of such a system.

Along with the above-listed functionality, the AT control solution provides a common User Interface (UI) that enables the user to fully parameterize test execution and customize all related tools (virtualization server, BTS, automated scripts) according to the project requirements.

At the maturity stage, the experience of earlier attempts at testing automation is taken into account, and special attention is paid to the scalability and expandable architecture of the ATCS itself.

IV. Main Challenges of Building An Automated Testing Control Solution

In order to remove manual activities from automated test execution, test engineers have a choice: whether to develop special configuration and command files, or attempt to create a special AT control program with a front-end interface that would send relevant instructions to the testing tools [5]. We focus on the latter, as this approach is more thoughtful and sustainable.

This section will cover the most common challenges that software companies are contending with while building automated testing solutions. These statements are based on our own professional experience, the experience of our colleagues and the results of our industry research.

With the view of studying certain problems of automated test execution, we conducted two online surveys. The first survey took place from May 10 to May 30, 2011 among the Russian-speaking IT community from all over the world. The link to the survey was placed at one of the most popular IT-specialized resource sites, Habrahabr.ru. The questionnaire consisted of 10 questions and gathered answers from 292 respondents [2]. The second survey was run among members of testing related groups on professional networking site LinkedIn.com from May 25, 2011 to June 4, 2011, and received 34 responses [3]. It was comprised of the same questions as the first survey and included an additional question (see Section IV-C below).

The objective of our surveys was to show that despite the use of automated tests there are manual routine operations a tester typically performs to have them executed.

A. *Incomplete Automation*

"When you start implementing automated tests, you will find that you are running the (supposedly automated) tests manually. Automating some part of test execution does not immediately give automatic testing [5]."

To assess the level of testing automation in their organizations, we asked a question in our survey: "During the automated testing, what are the operations that you still have to perform manually?" This particular question received 247 answers, and 45 respondents skipped it [2].

According to the survey, only 12.6% of respondents claim that in their organization all the operations related to AT are executed automatically, i.e., without the interference of an operator. As illustrated in Fig. 2, the most widespread tasks that a tester has to perform manually are submitting and closing bugs in the BTS (60.3% and 55.1%, respectively), launching automated tests (52.2%) and creating reports (44.5%). As the question allowed multiple answers, the total percentage exceeds 100 % [2].

On the one hand, these operations are monotonous and have a lower added value than, for example, the creation of new test cases – an alternative to investing the tester's time. On the other hand, they are time-consuming. For instance, in the case of data-driven testing, where the value of each particular output is important, a new bug must be submitted each time the test criteria are violated. On average, an experienced tester submits a defect into the bug-tracking system, including completing the assigned fields, in slightly more than a minute, and closing a fixed bug takes about 15 seconds[1]. Multiplied by the number of defects the tester has to process, the amount of wasted time may be considerable.

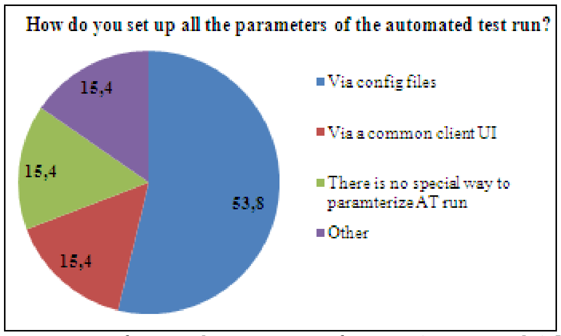


Figure 3. Configuring the parameters of AT run (survey results) [3].

[1] Measurements were done with the following properties:
1. Bug Tracking System (BTS): Microsoft Team Foundation Server (MS TFS), Mantis, Bugzilla.
2. Experimenters: 2 Testers (both 4 years of experience).
During the experiment 10 bugs were created with the following required fields:
MS TFS: Title, AssignTo, Iteration, Area, Tester, FoundIn, Severity.
Mantis: Category, Summary, Description, Platform, OS, Severity.
Bugzilla: Component, Version, Summary, Description, Severity, Assignee.
Opening BTS is also measured.

Another example is the set-up of the testing environment, which is carried out manually by 25.5% of our respondents [2]. The tester has to place a specific file into a specific directory before the automated test run. These actions are time-consuming, difficult to document and can be easily missed, resulting in flawed test results [5].

B. *System Scalability and Expandability*

In our interviews with peers, we found that oftentimes when a company develops a system for controlling automated testing, it focuses on the tools currently used without providing for system expandability. As a matter of fact, the solution being built for specific tools has important shortcomings. For instance, when the organization upgrades to a new version of the bug tracking system, or wants to add virtualization servers to the test lab, or introduces new types of automated tests created using a different framework, system integration and customization efforts will have a significant cost.

The outcome is the same when the crucial factor of system scalability is not taken into account. As product functionality increases over time, the number of automated tests increases as well, and there is a need for rational distribution of virtualization resources. The extension of the virtualization capabilities results in the rise of efforts to maintain the test automation system, and to manage a number of additional elements.

Therefore, such features as scalability and expandability have to be realized in the testing control solution's architecture in order to maximize its performance through the software life cycle.

C. *Absence of an Easy-to-use Control Tool (User Interface)*

In the majority of cases there is no single client interface for control and adjustment of the AT process, which negatively affects the overall performance. The settings of test execution are parameterized by means of config files (see Fig. 3) [3]. More often than not, the code of the configuration files is not subject to validation, resulting in an increase in human errors.

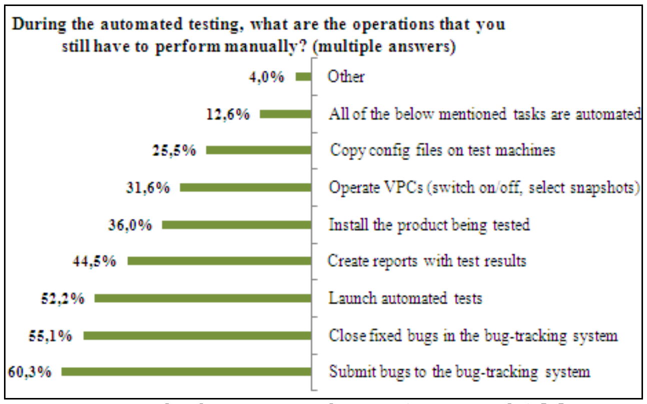


Figure 2. Manual tasks in automated testing (survey results) [2].

It is recommended to develop a front-end providing a "user interface that is independent of the automation tool used [5]. A common UI that enables the set-up of the automated test run without writing a single line of code augments the efficiency of software quality assurance. It helps reduce the learning curve, as the test engineer works with the single UI instead of interacting with several tools.

D. *Lack of Uniform Cumulative Reports*

In general, each test automation framework, such as HP QTP or IBM Rational Robot, generates reports in its own native format. In our practice we experienced firsthand the situation when a stakeholder (manager, customer) was not able to view test results because the corresponding tool was not installed on his/her computer. Even if any of these frameworks possesses an export feature, they have to handle a stack of separate lengthy records.

A testing automation solution should provide for uniform cumulative reports, meaning that a single summary report is based on the results of a batch of test cases and is presented in a structured and easily readable form. Furthermore, up-to-date results should be available and accessible at any moments of test execution, and the storage of reports history should be enabled.

E. *Uninterrupted Operation*

An automated testing process resembles a conveyor belt. At the entry point, we have new builds of the system under test, and at the output, found defects and reports. To ensure the continuity of operation and system stability, it is necessary to develop mechanisms preventing system hang-up. For instance, if a test has an error, it will be run endlessly, keeping a virtual machine's resources busy and preventing the execution of other queued tests. Therefore, it is useful to implement the "timeout kill" algorithm to ensure the system's fault tolerance.

F. *Insufficient or Lacking System log*

Deficient system logging hampers the debugging process, which makes an ATCS non-transparent and its activities hardly traceable. Therefore, when developing a system for automated testing control, it is crucial to enable the logging of all system components, including the events of automated tests, virtual machines, defect management system, reports, etc. These measures help minimize the time needed for debugging and increase the efficiency of software quality assurance and validation.

V. Architecture Proposal of Automated Testing Control Solution

In this section, we describe an efficient approach to bring automated test execution to the highest maturity level. We present a working archetype of the software system that controls automated tests and is independent of testing tools used. The proposed solution eliminates routine manual operations from the test execution process.

A. *Integration of Testing Tools*

The approach we recommend consists of building a coherent and comprehensive software solution which independently controls all operations related to AT – from launching tests and operating virtual machines to submitting bugs and generating reports. The solution, as Fig. 4 suggests, is based on the integration of software tools engaged in AT, namely the file server, the build machine, the versioning control system, the virtualization server, the bug tracking system, and automated tests themselves.

In order to develop such an integrator, first one needs to analyze the tester's interaction with all the above-mentioned tools. The second step is to examine the APIs (Application Programming Interfaces) of each tool. The final stage is the development of a solution that integrates all these software tools under a common UI, via which the tester can easily and quickly adjust the automated testing control system according to the requirements and processes established in the organization.

In other words, instead of customizing and configuring each tool separately (virtualization server, BTS, automated tests, etc.), the tester will be able to adjust all settings via a single easy-to-use UI.

The prototype of the described automated testing control system was developed and successfully deployed by Applied Systems Ltd. The program architecture consists of three modules:

1) *Automated Test Manager (ATManager)*

ATManager is a complex service that controls the whole AT process and assures communication among all elements in the system. It plays a central part in the functioning of the test automation solution and works using the algorithms described below.

When a new project is created, the tester (operator) presets the ATCS for verifying a specific build branch: adds tests into the system, groups them into test runs, etc. Once this is done, ATManager takes over and probes every new build in automatic mode.

1. ATManager monitors the state of the build machine via its API. If the new build is completed successfully, ATManager is notified and starts the testing procedure. Each build can have several test runs configured to verify it. Different test runs can be executed simultaneously on different machines.
2. ATManager finds an appropriate test machine (VM or physical PC). Each test run has a set of virtual machines on which they can be executed.

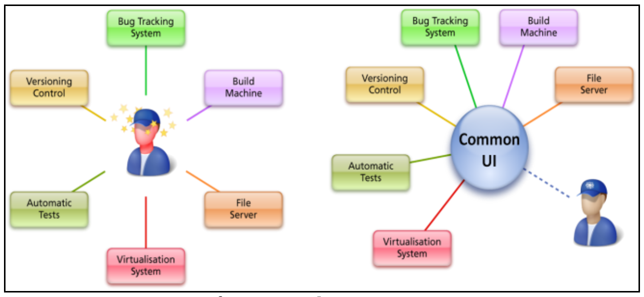


Figure 4. Integration of testing tools.

3. ATManager starts a VM (in the case of using virtualization during testing) via the API of the corresponding virtualization server. It chooses appropriate machines from the least busy virtualization server.
4. ATLauncher is initialized.
5. ATManager deploys build binaries and automated tests from the file server or version control system on the VM. Then it configures the environment on the test machine. ATLauncher launches tests.

…Automated tests are executed…

6. ATManager sends test results to the File Server.
7. ATManager submits/updates defects to the BTS, closes fixed bugs if these actions were allowed by the tester. ATManager service fills in the required fields in the BTS (e.g., Title, Tester, Product, Assigned to, etc.) using its API (see Fig. 5 for illustration).

2) Agent for Launching Automated Tests (ATLauncher)

ATLauncher is a console application installed on each testing machine. It is a small service that does not impact the performance of the host system.

The main functions of the module are:

- presetting the testing environment (i.e., copy configuration files, install the software under test);
- launching various types of tests (using the APIs of frameworks in which they were developed);
- processing and converting the test results, etc.

The module architecture must be expandable and allow the addition of new features.

ATLauncher starts working after ATManager has started a VM (in the case of using virtualization) and copied all required files, including automated test scripts and config files. The XML file created by the control module ATManager contains the description of tests and usage instructions. As soon as the tests are finished, ATLauncher creates a special results file to notify ATManager about the completion of its task.

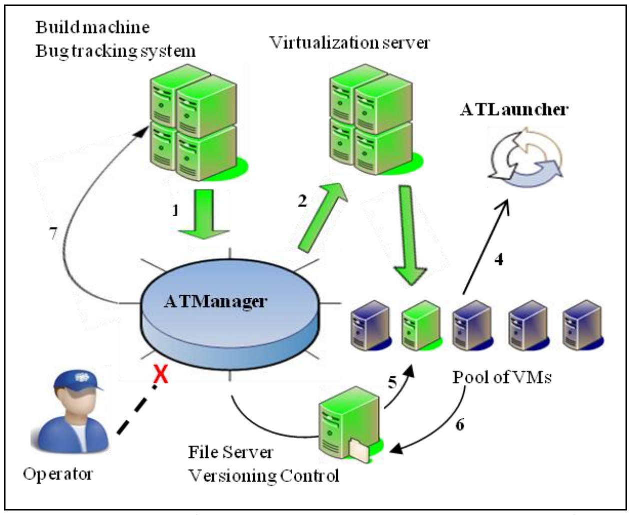


Figure 5. Scheme of communication between the components of the ATCS.

3) Control Panel

The user interface is represented by the control panel. It is a client application which facilitates the interaction between the tester and the ATCS, providing the tools necessary to configure and manage the test run for an application under test.

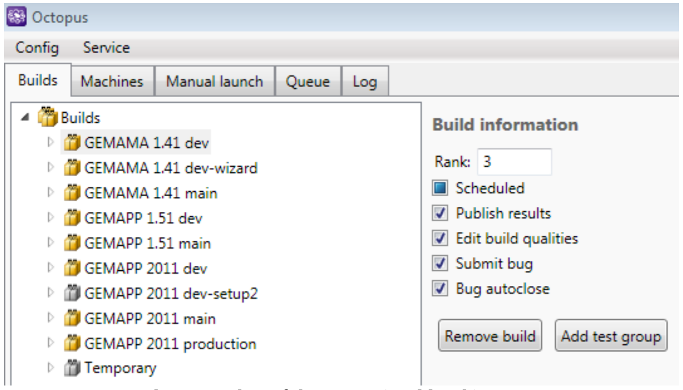
The UI allows the user to:

1. Specify the tests to be run on each build, assign priority to the build branch, schedule test launch on event (issue of a new build) or on schedule; choose defect tracking options (Fig. 6).
2. Allocate the sets of valid machines for each test run, assign tests for execution on a particular real or VMs and their snapshots.
3. Manually launch tests on a specific build, interrupt test execution.
4. View ATManager's logs.
5. Monitor the testing queue in real-time (Fig. 7).

In the time of ever-increasing mobility, it is useful to provide access to the control panel from the desktop as well as a web interface.

B. Distribution of Functionality Among Components

While creating a solution to control an automated testing cycle it is necessary to distribute the functionality of your future system among its components.



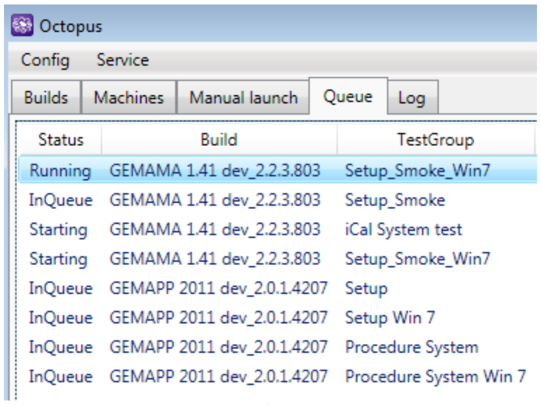
Builds
- GEMAMA 1.41 dev
- GEMAMA 1.41 dev-wizard
- GEMAMA 1.41 main
- GEMAPP 1.51 dev
- GEMAPP 1.51 main
- GEMAPP 2011 dev
- GEMAPP 2011 dev-setup2
- GEMAPP 2011 main
- GEMAPP 2011 production
- Temporary

Build information

Rank: 3

- [ ] Scheduled
- [x] Publish results
- [x] Edit build qualities
- [x] Submit bug
- [x] Bug autoclose

Figure 7. Sample screenshot of the ATCS(Builds tab).



Status	Build	TestGroup
Running	GEMAMA 1.41 dev_2.2.3.803	Setup_Smoke_Win7
InQueue	GEMAMA 1.41 dev_2.2.3.803	Setup_Smoke
Starting	GEMAMA 1.41 dev_2.2.3.803	iCal System test
Starting	GEMAMA 1.41 dev_2.2.3.803	Setup_Smoke_Win7
InQueue	GEMAPP 2011 dev_2.0.1.4207	Setup
InQueue	GEMAPP 2011 dev_2.0.1.4207	Setup Win 7
InQueue	GEMAPP 2011 dev_2.0.1.4207	Procedure System
InQueue	GEMAPP 2011 dev_2.0.1.4207	Procedure System Win 7

Figure 7. Sample screenshot of the ATCS(Queue tab).

In Table 1, we suggest possible options to distribute basic functionality among ATM, ATL and the client UI. To coordinate the work of all these components, one needs to develop a large set of algorithms and solutions.

VI. Conclusion and Future Work

One of the most serious problems facing software development companies today is the lack of resources for regular and comprehensive regression testing.

The most obvious and popular solution is implementing automated testing. However, despite the abundance of tools for testing automation, this endeavor presents many challenges, especially in the case of testing desktop applications. In the first place, the word "automated" does not mean, as one might be led to believe, that operations are handled without human interaction. In fact, in the process of automated test execution – *Configure the testing environment → Start the virtual test machine → Launch tests → Execute tests → Submit bugs to the bug-tracking system → Generate test reports* – only a few operations, besides the test execution itself, are automated. In addition to the fact that "non-automated" activities are time-consuming and inefficient, they also leave room for human error.

To meet these challenges, some companies developing complex software are trying to create a solution that would control the whole AT process from A to Z without the participation of an operator. Only 12.6% succeed [2].

TABLE 1. Distribution of Basic Functionality Among Components of ATCS

Functionality	**Module**		***Control via common UI***
	AT-Manager	***AT-Launcher***	
1. Operate virtual machines (VM) - start/shut down VM - add VM and snapshots to the system - group VMs	+	-	+
2. Launch automated tests	-	+	+
3. Submit/close bugs in the BTS	+	-	+
4. Generate a uniform cumulative report	+	-	+
5. Convert results into a single easy-to-interpret format	+	-	-
6. Copy tests, config files, product setup files to the testing machine	+	-	+
7. Install the tested product	-	+	+
8. Log all system events	+	+	-
9. Abort text execution	+	+	+

In this paper, we have described an efficient and innovative approach to automating test execution based on the integration of all testing tools under a common UI. We also provided practical advice on how to develop such an AT control solution, proposed the system architecture, defined the key functionality of its components and schematized the communication among them.

In the future, we plan to assess the costs and benefits of implementing an AT control solution into a company's QA management system.

Acknowledgment

We would like to express gratitude to Ivan Bachtin, who did a wonderful job developing the prototype of the automated testing control system we refer to in this paper, and to Sergei Usovich, who provided financial support and shared our enthusiasm. We also thank our friends and colleagues who shared their experience and insights and contributed to this work, namely Maks Shatokhin, Alexander Abramov, and Dmitry Romanovich.

In fact, a few hundred people contributed to the progress of this paper in different ways, and we are especially thankful to the respondents to our online surveys, which provided a solid support to our statements and arguments.

References

[1] D. Mosley, B. Posey, "Just Enough Test Automation," Prentice Hall, 2002, pp. 12-14.

[2] Online survey "Problems of automated desktop software testing" by Applied Systems Ltd. via Habrahabr.ru, https://www.surveymonkey.com/s/automated_testing_problems 30.05.2011.

[3] Online survey "Challenges of automated software testing" by Applied Systems Ltd. via LinkedIn.com, https://www.surveymonkey.com/s/automated_testing, 30.05.2011.

[4] E. Dustin, J. Rashka, and J. Paul, "Automated Software Testing: Introduction, Management, and Performance," Addison-Wesley Professional, 1999, pp. 38- 45.

[5] M. Fewster, D. Graham, "Software Test Automation: Effective use of test execution tools," Addison-Wesley Professional, 1999, pages: 3, 62, 246, 329.

[6] E. Dustin, T. Garrett, "Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality," Addison-Wesley Professional, 2009, pp. 192-204.

[7] T. Garrett, "Useful Automated Software Testing Metrics," http://idtus.com/img/UsefulAutomatedTestingMetrics.pdf, 21.07.2011.

[8] M. Krause, "A Maturity Model for Automated Software Testing," Medical Device and Diagnostic Industry Magazine, December 1994.

[9] I. Burnstein, T. Suwanassart, and C. Carlson, "The Development of a Testing Maturity Model," Proc. Ninth International Quality Week Conference, San Francisco: The Software Research Institute, 1996.

Detecting Equivalent Mutants by Means of Constraint Systems

Simona Nica, Mihai Nica and Franz Wotawa

Technical University of Graz, Institute for Software Technology
Inffeldgasse 16B/II, A-8010 Graz, Austria
{snica, mnica, wotawa}@ist.tugraz.at

***Abstract*—Mutation testing has been used along the research community as an efficient method to evaluate the process of software testing, i.e., the quality of the test suite. One major drawback is represented by the equivalent mutant problem. Through this current research we aim to come with a reliable solution to this problem and improve the available test suite pool. We do this by combining the mutation testing procedure together with a constraint satisfaction paradigm and the concept of distinguishing test cases. Mutation testing has been seen, in most of the cases, as a measure for evaluating the quality of a user's test suite. But, also mutation testing can be of great help in the test case generation process. By means of a constraint system we generate test scenarios able to distinguish between two different versions of a program. We start from the hypothesis that when our constraint system is not able to find any solution it might be the case that two equivalent mutants were encountered. The first empirical results, i.e. an increased mutation score, encourage us to further apply the strategy on medium size applications.**

Keywords-Mutation Testing; Equivalent Mutants; Mutation Score; Constraint Satisfaction Problem; Distinguishing Test Case.

I. Introduction

Mutation testing has been intensively used in a large number of experiments as an efficient way to detect the quality of a program's test suite [1]. It is a fault based technique that makes use of a well determined set of faults for measuring the efficiency of the test suite. In mutation testing the original program is slightly changed using mutation operators and the resulting mutant is executed using the test suite. If there is at least one failing test run, the mutant is said to be detected or killed. In mutation testing mutants that are not killed are alive. A test suite is said to be more effective if it has the capability to detect more mutants. The efficiency of a test suite in mutation testing is measured using the *mutation score*. The mutation score is equivalent to the number of mutants detected divided by the overall number of of non equivalent mutants (a mutant is said to be equivalent if there is not a test case which can distinguish the output of the mutant from the output of the original program). In the ideal case the mutation score is 1 and all mutants are successfully detected. Mutation analysis is a good metric for measuring the coverage levels achieved [2]. In the work presented in this paper, we are using mutations not only for measuring the efficiency but also for improving the quality of the test suite. The idea is to generate new test cases in case of mutants that cannot be detected. The proposed technique is based on the constraint representation of programs [3], [4] and on the concept of distinguishing test cases [5]. We convert both the program and its alive-mutants to a constraint satisfaction problem (CSP) [6] and ask the constraint solver to search for an input such that the two programs differ by at least one output value (computation of a distinguishing test case). When receiving such an input we are able to discriminate the mutant and the original program using the generated test case. An input that allows to discriminate two programs is called a *distinguishing test case*.

However, sometimes it might happen that a mutation over the original program will not change the semantics of the program, making thus hard to detect the change by a test case. This is one important issue which must be considered when generating the mutants. In the literature this is denoted as the equivalent mutant problem. Although several techniques are available in order to solve this problem [7], [8], we do not have a general solution. Therefore, in the first phase, we propose an algorithm to detect and reduce the equivalent mutants and then apply the distinguishing test cases algorithm in order to improve the test suite of a given application, i.e. improve the mutation score.

The goal is to clarify the research question whether it is always possible to increase the mutation score of a test suite from $x\%$, e.g., 70%, to 100%, based on the method of computing distinguishing test cases from alive-mutants and eliminating the equivalent ones. Our hypothesis in this respect is that a mutation score close to 100% can be achieved when using our proposed technique. In some initial experiments we observed increases of the mutation score even from 42% to 100%. However, in these experiments we only used small-size programs for testing the algorithm. Hence, the initial experiments confirmed our hypothesis and further more sophisticated experiments have to be carried

The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT) and the Austrian Science Fund (FWF).

Authors are listed in alphabetical order.

out.

In what follows, we will give the basic definitions and then describe the proposed algorithm.

II. Basic Definitions

In order to have an accurate understanding of the algorithms described later in the paper, we present the basic definitions which we will use throughout our paper. We will explain what a mutation is, what we understand by equivalent mutant and what a constraint system is.

***Definition** 1:* **[Test Case]** A test case for a program Π is a set (I, O) where I is the input variable environment specifying the values of all input variables used in Π, and O the output variable environment, which does not necessarily specifies the values for all output variables.

A test case is a *failing test case* if and only if the output environment computed from the program Π when executed on input I is not consistent with the expected environment O. Otherwise, we say that the test case is a *passing test case*. If a test case is a failing (passing) test case, we also say that the program fails (passes) executing the test case.

***Definition** 2:* **[Test Suite]** A test suite TS for a program Π is a set of test cases of Π.

***Definition** 3:* **[Constraint Satisfaction Problem (CSP)]** A constraint satisfaction problem is a tuple (V, D, CO) where V is a set of variables defined over a set of domains D connected to each other by a set of arithmetic and boolean relations, called constraints CO. A solution for a CSP represents a valid instantiation of the variables V with values from D such that none of the constraints from CO is violated.

The variables from the CSP system do not necessarily need to be the variables used in the program.

***Definition** 4:* **[Mutant]** Given a program Π and a statement $S_\Pi \in \Pi$. Further let S'_Π be a statement that results from S_Π when applying changes like modifying the operator or a variable. We call the program Π', which we obtain when replacing S_Π with S'_Π , the mutant of program Π with respect to statement S_Π.

***Definition** 5:* **[Equivalent Mutant]** Given a program $\Pi \in \mathcal{L}$ and one of its mutant Π', we say that Π' is an equivalent mutant if the mutation that differentiates Π from Π' does not change the semantic of Π.

For a better understanding, we illustrate our definition with the example from Figures 1 and 2. Over the original version of the program from 1 we apply the relational operator replacement $\geq$.

***Definition** 6:* **[Distinguishing Test Case]** Given a program Π and one of its mutant Π', a distinguishing test case for program Π and its mutant Π' is a tuple $(I, \emptyset)$ such that for the input value I the output value of program Π differs from the output value of program Π'.

```
int a, b;
int compute;
if (a == b)
  compute = a;
else
  compute = (a + b)/2;
System.out.println(compute);
```

Figure 1. Original Program

```
int a, b;
int compute;
if (a >= b)
compute = a;
 else
  compute = (a + b)/2;
System.out.println(compute);
```

Figure 2. Equivalent Mutant for ROR

III. Research Strategy

Mutation testing is used mainly to determine the effectiveness of the given test suite by making use of the mutation score metric [9]. The idea of using mutation testing also for test case generation is not new. In [10], the authors use model based mutation testing in order to obtain distinguishing test cases from contract mutations. In [5], there are used distinguishing test cases obtained from mutants to reduce the number of diagnoses in case of fault localization. Mutation can also be used to indicate possible fixes of faulty programs as suggested in [11]. Moreover, the use of constraints for test case generation is also not new. In [12], the authors propose a method that makes use of the constraint systems to generate test cases. What distinguishes our work from the previous one is the combination of program mutation and constraint solving techniques in order to improve the mutation score of the test suites and, moreover, to help detecting the equivalent mutants [13].

Moreover, we try to determine an efficient method for eliminating the equivalent mutants. In what follows, we will describe first the algorithm we use to detect and remove the equivalent mutants from the set of generated mutants, and then the algorithm for improving the mutation score of test suite.

In our research, we make use of an extended version of the MuJava tool [14], [15] for computing the mutants and the MINION constraint solver [16] for obtaining the distinguishing test cases.

First, we define an algorithm which will translate the original program into a constraint system. We will call this algorithm along the research experiment. It receives as input the original program and it gives as output the constraint system.

```
Algorithm Transform_To_CSP $(\Pi)$
1) Eliminate all loops from the original program by replacing them with a bounded number of nested conditional statements,
2) Convert $\Pi$ to its equivalent SSA (static single assignment form) representation,
3) Convert $SSA_\Pi$ into its corresponding constraint representation system.
```

For the elimination phase, the algorithm receives as input the original program Π and the set of generated mutants M_Π and offers at the output the new set of mutants.

```
Algorithm Eliminate_Equivalent_Mutants $(\Pi, M_\Pi)$
1) Call Transform_To_CSP $(\Pi)$ and obtain the constraint system $CON_\Pi$
2) For each $\Pi_i$ from $M_\Pi$
    a) Call Transform_To_CSP $(\Pi_i)$ and get the mutant constraint system $CON_{\Pi_i}$
    b) Create the constraint system $CS$, corresponding to $CON_\Pi \wedge CON_{\Pi_i}$, in order to apply the distinguishing test case restriction to the entire constraint system;
    c) Add the same inputs - different outputs constraints, i.e., $I(CON_{\Pi_i}) = I(CON_\Pi)$ and $O(CON_{\Pi_i}) \neq O(CON_\Pi)$ to the set of constraints $CS$.
    d) Solve the constraint system $CS$.
    e) if no solution is found, then do:
        i) Equivalent mutant detected
        ii) Remove mutant $\Pi_i$ from $M_\Pi$
```

The above algorithm will always end either when one or several solutions are found or when the constraint system is not able to detect any solution. The condition *same input - different outputs* is first used in our research in order to help us detect an equivalent mutant. The experiments conducted have demonstrated that when the constraint solver is not able to offer at least one solution, the two programs taken into consideration are semantically *equivalent*.

Now, we present the method for improving the test suite of a given program. The algorithm takes as input the program Π and the test suite TS, and delivers as output a test suite that must have a higher mutation score than the original one.

```
Algorithm Generate_Test_Cases $(\Pi, TS)$
1) For the program $\Pi$ generate the finite set of mutants $M_\Pi$.
2) $M_\Pi$ = Eliminate_Equivalent_Mutants $(\Pi, M_\Pi)$
3) Run the original test suite $TS$ against the set of mutants $M_\Pi$ and, compute the mutation score $\mu = \frac{Mutants_{Killed}}{Mutants_{Total}}$ where $Mutants_{Killed}$ is the number of killed mutants, and $Mutants_{Total}$ represents the total number of mutants.
4) If the mutation score $\mu$ is larger than a predefined value, return $TS$ as result. In this case no improvement is necessary.
5) Otherwise, for each $\Pi_i$ from $M_\Pi$
    a) Call Transform_To_CSP $(\Pi_i)$ in order to convert the original program and the alive mutants into their CSP representation (for more information concerning program conversion to its constraint representation we refer the interested reader to [4])
6) Let $CON_\Pi$ be the constraint representation of the original, bug-free, program.
7) For every constraint representation $\Pi_{DS_i}$ of the available set of mutants $\Pi_{DS}$, $i = 1, ..., |\Pi_{DS}|$ do:
    a) Let $CS$ be the set of constraints comprising the constraints from $CON_\Pi$ and $\Pi_{DS_i}$.
    b) Add the same inputs - different outputs constraints, i.e., $I(\Pi_{DS_i}) = I(CON_\Pi)$ and $O(\Pi_{DS_i}) \neq O(CON_\Pi)$ to the set of constraints $CS$.
    c) Solve the constraint system $CS$.
    d) if a solution is found, then do:
        i) Let $T'$ denote the valid test case that kills mutant $\Pi_{DS_i}$.
        ii) Add $T'$ to the test suite $TS$;
        iii) Run $T'$ against the set of mutants $\Pi_{DS}$ of program $\Pi$ and eliminate $\Pi'_{DS_i}$ from $\Pi_{DS}$ if $\Pi'_{DS_i}$ fails on $T'$.
8) Compute the mutation score $\mu$ and return $TS$ as result.
```

Our research experiment was run over a small set of simple Java programs (no more than 200 lines of code), e.g, bubble sort, arithmetic operations, and some of the classes belonging to HTML Parser project [17] - a Java library used to parse HTML. Only small deviations, i.e., mutants that are close to the original program, were taken into consideration. Up do now we did not benefit from a significant test pool, but we were able to obtain a higher mutation score with a small number of generated distinguishing test cases and a smaller number of mutants. In order to demonstrate the practicability of our approach, we intend to substantially extend the empirical results based on larger programs with a variety of test suites.

In Table I, we summarize the first empirical results of our approach. By **LOC** we denote the lines of code, by $\mathbf{Line_{Cov}}$ we show the line coverage. For each class we record the initial mutation score, $\mathbf{MS_{Init}}$, resulted from the normal mutation testing procedure, and then, after applying our algorithm, we compute the new mutation score

Class	LOC	$\mathbf{Line_{Cov}}$	$\mathbf{MS_{Init}}$	$\mathbf{MS_{DTC}}$
tagTests.AppletTagTest	96	52.00%	27.41%	65.00%
tagTests.BaseHrefTagTest	13	23.00%	18.10%	54.13%
tagTests.BodyTagTest	7	86.00%	79.00%	84.30%
tagTests.CompositeTagTest	156	27.00%	17.56%	51.12%
tagTests.FormTagTest	46	11.00%	6.18%	19.23%
tagTests.LinkTagTest	58	43.00%	31.33%	65.34%
DivATC	21	100.00%	67.66%	100.00%
SumATC	18	100.00%	41.87%	100.00%
BubbleSort	43	99.97%	56.40%	79.10%

Table I
MUTATION SCORE WITH DISTINGUISHING TEST CASES

$\mathbf{MS_{DTC}}$, not taking into account the equivalent mutants.

The strategy is prone to some limitations, connected to the mutation testing tool and the constraint solver we use. The MINION solver does not support object-oriented constructs. Concerning the mutations we produce, we are not able to mutate constant values, nor to add or remove statements.

IV. CONCLUSION

In this paper, we aim at improving the quality (given as the mutation score) of a program's test suite. We achieve this by generating distinguishing test cases for extending the available test suite, and also by reducing the number of equivalent mutants. A distinguishing test cases is a test case that allows for distinguishing a program from its mutant using the same input. When adding this test case to the test suite, the mutation score of the new test suite has to increase, assuming a mutant that is not equivalent to the original program.

Up to now, the obtained empirical results support the claim that our approach improves test suites. However, we further strengthen the empirical results and aim to test our algorithm on medium scale applications.

REFERENCES

[1] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in *IEEE Transactions of Software Engineering*, vol. PP, no. 99, Paris, France, 2010, p. 1.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," in *IEEE Transactions on Software Engineering*, September 2006, pp. 608–624.

[3] R. Ceballos, M. Nica, J. Weber, and F. Wotawa, "On the complexity of program debugging using constraints for modeling the program's syntax and semantics," in *Proc. Conference of the Spanish Association for Artificial Intelligence (CAEPIA)*, Seville, Spain, 2009, pp. 22–31.

[4] M. Nica, J. Weber, and F. Wotawa, "How to debug sequential code by means of constraint representation," in *International Workshop on Principles of Diagnosis (DX-08)*, Leura, Australia.

[5] F. Wotawa, M. Nica, and B. K. Aichernig, "Generating Distinguishing Tests using the MINION Constraint Solver," in *CSTVA 2010: Proceedings of the 2nd Workshop on Constraints for Testing, Verification and Analysis*, Paris, France, 2010, pp. 325–330.

[6] R. Dechter, *Constraint Processing*. The Morgan Kaufmann Series in Artificial Intelligence, 2003.

[7] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," in *Software Testing, Verification, and Reliability*, 1994, pp. 131–154.

[8] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," in *ICST '10: Third International Conference on Software Testing, Verification and Validation*. Paris, France: IEEE Computer Society, April 2010, pp. 45–54.

[9] J. H. Andrews, L. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" in *Proceedings of IEEE International Conference on Software Engineering*, St. Louis, MO, USA, May 2005, pp. 402–411.

[10] W. Krenn and B. K. Aichernig, "Test Case Generation by Contract Mutation in Spec #," in *Electronic Notes in Theoretical Computer Science*, 2009, pp. 71–86.

[11] V. Debroy and W. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, Paris, France, 2010, pp. 65–74.

[12] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," in *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software testing and analysis*, Clearwater Beach, Florida, United States, 1998, pp. 53–62.

[13] B. J. M. Grün, D. Schuler, and A. Zeller, "The Impact of Equivalent Mutants," in *IEEE International Conference on Software Testing, Verification, and Validation Workshops*, Denver, USA, 2009, pp. 192–199.

[14] Y. Ma, J. Offutt, and Y. Kwon, "Mujava : An automated class mutation system," in *Software Testing, Verification and Reliability* , 2005, pp. 97–133.

[15] S. Nica and B. Peischl, "Challenges in Applying Mutation Analysis on EJB-based Business Applications," in *Proceedings of Metrikon 2009*, Kaiserslautern, Germany, November 2009.

[16] I. Gent, C. Jefferson, and I. Miguel, "Minion: A fast, scalable, constraint solver," in *17th European Conference on Artificial Intelligence ECAI-06*, Trento, Italy, 2006, pp. 98–102.

[17] HTML Parser, "http://htmlparser.sourceforge.net/," 2011.

Answer-Set Programming as a new Approach to Event-Sequence Testing

Esra Erdem[1], Katsumi Inoue[2], Johannes Oetsch[3], Jörg Pührer[3], Hans Tompits[3], Cemal Yilmaz[1]

[1]*Sabanci University, Faculty of Engineering and Natural Sciences,*
Orhanli, Tuzla, Istanbul 34956, Turkey
Email: {esraerdem,cyilmaz}@sabanciuniv.edu

[2]*National Institute of Informatics,*
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
Email: ki@nii.ac.jp

[3]*Technische Universität Wien, Institut für Informationssysteme 184/3,*
Favoritenstraße 9-11, A-1040 Vienna, Austria
Email: {oetsch,puehrer,tompits}@kr.tuwien.ac.at

***Abstract*—In many applications, faults are triggered by events that occur in a particular order. Based on the assumption that most bugs are caused by the interaction of a low number of events, Kuhn et al. recently introduced *sequence covering arrays* (SCAs) as suitable designs for event sequence testing. In practice, directly applying SCAs for testing is often impaired by additional constraints, and SCAs have to be adapted to fit application-specific needs. Modifying precomputed SCAs to account for problem variations can be problematic, if not impossible, and developing dedicated algorithms is costly. In this paper, we propose *answer-set programming* (ASP), a well-known knowledge-representation formalism from the area of artificial intelligence based on logic programming, as a declarative paradigm for computing SCAs. Our approach allows to concisely state complex coverage criteria in an *elaboration tolerant* way, i.e., small variations of a problem specification require only small modifications of the ASP representation.**

***Keywords*-event-sequence testing; combinatorial interaction testing; answer-set programming.**

I. Introduction

In many applications, faults only show up if events occur in a certain order. An example are atomicity violations in multi-threaded applications where a pair of shared memory accesses of one thread is interleaved with an unfortunate access of another thread. Testing such applications thus requires exercising *event sequences*. Since the number of event sequences is factorial in the number of events, exhaustive testing is infeasible in general. If we assume that bugs are triggered by the interaction of only a low number of events—this is empirically supported by respective bug reports—, testing costs can be reduced drastically without sacrificing much fault-detection potential by using suitable combinatorial designs [1], [2]. To this end, Kuhn et al. [3], [4] introduced *sequence covering arrays* (SCAs) for combinatorial event sequence testing. An SCA is an array of permutations of events such that any t events, possibly interleaved with other events, will be tested in every t-way order at least once. SCAs are relevant in scenarios where the order of events is decisive, like testing of user-interfaces, dynamic web applications, method calls for unit-testing, or multi-threaded programs.

In practice, a direct application of SCAs for testing is often impaired by additional constraints on the order of events. Also, the conditions that identify the sequences that should be covered can vary and often involve quite complex definitions. For example, to test thread interleavings, one could require to test all sequences such that one variable is written by one thread and subsequently read by another thread such that there is no write operation between them [5], [6].

One approach to address such considerations is to accordingly modify precomputed SCAs as exemplified by Kuhn et al. [3], [4]. This means that any test sequence which, e.g., violates some ordering constraints has to be removed from the SCA. To maintain coverage, removed sequences have to be replaced by permutations thereof that comply to the problem specific requirements. This is not always possible in a straightforward way and can result in a considerable and in principle avoidable overhead regarding the size of arrays. On the other hand, developing and maintaining dedicated algorithms to compute variations of SCAs usually comes with high costs and is not preferable if requirements change over time or one wants to experiment with different designs.

We propose to use *answer-set programming* (ASP) [7] for computing SCAs and variations thereof. ASP is a genuine declarative programming paradigm where a problem is encoded by means of a logic program such that the solutions of a problem correspond to the models, called *answer sets*, of the program. On the one hand, as an expressive high-level specification language, it allows to state complex coverage criteria, involving constraints and complex, possibly recursive, definitions, in a concise and *elaboration-tolerant* way, i.e., small variations in a problem specification require only small modifications of the program representation. On the other hand, SCAs can be efficiently computed through highly optimised ASP solvers [8]. Since it requires only little effort to state quite complex coverage conditions in ASP, a tester is able to rapidly specify different versions of SCAs.

This paper is organised as follows. In Section II, we review SCAs and ASP. Then, we show how SCAs can be generated using ASP in Section III. We present improved, sometimes optimal, upper bounds regarding the size of many SCAs. We furthermore present a greedy algorithm, based on ASP, for computing larger SCAs. In Section IV, we turn towards a real-world example described by Kuhn et al. [3], [4]. We discuss how the basic ASP encoding from Section III can be refined to take different constraints and problem variations into account. Finally, we discuss related work in Section V and conclude in Section VI.

II. Preliminaries

In this section, we review the formal definition of SCAs and give a brief background on ASP.

A. Sequence Covering Arrays (SCAs)

SCAs, introduced by Kuhn et al. [3], [4], are combinatorial designs related to covering arrays. While covering arrays require that each t-way combination of parameters occurs at least once in a test case for some fixed t, SCAs take the order of events into account and require that each t-sequence of events is tested in at least one test sequence in that order, where a *t-sequence* over a set S of symbols is a t-tuple of pairwise distinct elements of S. Following Kuhn et al. [3], [4], we formally define SCAs as follows.

Definition 1: A *sequence covering array* (SCA) with parameters n, S, and t, or an (n, S, t)-SCA for short, is an $n \times |S|$ matrix M of symbols from a finite set S of symbols such that (i) each row of M is a permutation of S and (ii) for each t-sequence $\sigma = (s_1, s_2, \ldots, s_t)$ over S, there is at least one row $\varrho = (a_{i1}, \ldots, a_{i|S|})$ in M such that σ is a subsequence of ϱ.
We say that an (n, S, t)-SCA is of *strength* t and of *size* n. The *sequence covering array number* $\mathrm{SCAN}(S, t)$ is the smallest n such that an (n, S, t)-SCA exists. An (n, S, t)-SCA is *optimal* if $\mathrm{SCAN}(S, t) = n$. We will also denote an $(n, \{1, \ldots, s\}, t)$-SCA as an (n, s, t)-SCA with $\mathrm{SCAN}(s, t)$ for brevity.

For illustration, the following matrix M constitutes an optimal $(7, 5, 3)$-SCA:

$$M = \begin{pmatrix} 5 & 2 & 3 & 1 & 4 \\ 3 & 2 & 5 & 4 & 1 \\ 1 & 5 & 4 & 3 & 2 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 2 & 5 & 1 & 3 \\ 2 & 4 & 3 & 1 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}.$$

Each of the 7 rows is a permutation of the set $S = \{1, \ldots, 5\}$ and each 3-sequence over S is covered by at least one row. For instance, the 3-sequence $(5, 3, 4)$ is covered by the first row of M. Note that there are $5 \cdot 4 \cdot 3 = 40$ such 3-sequences.

A collection of precomputed SCAs of strength 3 and 4, involving 5 to 80 events, is available online [9].These SCAs were computed using a simple greedy algorithm introduced by Kuhn et al. [3], [4]. Note that this algorithm is the only approach for computing SCAs implemented so far. To compute a t-strength SCA for a set S of events, this algorithm iteratively computes single rows of the SCA: It computes a fixed number of permutations of S. Then, it selects the permutation π that obtains maximal coverage of previously uncovered t-sequences as the next row of the SCA. After that, π in reverse order, π', is added. Adding π' is justified because π' always covers the same number of previously uncovered t-sequences as π [4]. This procedure is iterated until all t-sequences are covered.

One downside of this greedy algorithm is that additional constraints on the order of events arising from the requirements of different test scenarios are hard to incorporate. To overcome this shortcoming, we use ASP in what follows as a declarative tool to compute SCAs and demonstrate that quite complex constraints can be incorporated into a solution in a concise and elaboration-tolerant way, and with ease.

B. Answer-Set Programming (ASP)

ASP [7] is a relatively new declarative programming paradigm. The underlying idea of ASP is to declaratively represent a computational problem as a logic program whose models, called "answer sets", correspond to the solutions, and to find the answer sets for that program using an ASP solver. Due to the expressiveness of ASP that allows to represent, for instance, aggregates and recursive definitions, and due to the continuous improvements of the efficiency of ASP solvers, such as `clasp` [10], we argue that ASP can efficiently and effectively be used to compute SCAs. Indeed, ASP has been used in a wide range of applications from different fields, such as semantic-web reasoning, systems biology, planning, diagnosis, information integration, configuration, multi-agent systems, cladistics, and super optimisation. For a comprehensive introduction to ASP, we refer to the textbook by Baral [7].

We recapitulate the basic elements of ASP in the following. An *answer-set program* is a finite set of rules of the form

$$a_0 :- \ a_1, \ldots, a_m, \mathrm{not}\ a_{m+1}, \ldots, \mathrm{not}\ a_n, \qquad (1)$$

where $n \geq m \geq 0$, a_0 is a propositional atom or $\bot$, and all $a_1, \ldots, a_n$ are propositional atoms; the symbol "not" denotes *default negation.* If $a_0 = \bot$, then Rule (1) is a *constraint* (in which case a_0 is usually omitted). The intuitive reading of a rule of form (1) is that whenever $a_1, \ldots, a_m$ are known to be true and there is no evidence for any of the default negated atoms $a_{m+1}, \ldots, a_n$ to be true, then a_0 has to be true as well. Note that $\bot$ can never become true.

An *answer set* for a program is defined following Gelfond and Lifschitz [11]. An *interpretation* I is a finite set of propositional atoms. An atom a is *true* under I if $a \in I$, and *false* otherwise. A rule r of form (1) is true under I if $\{a_1, \dots, a_m\} \subseteq I$ and $\{a_{m+1}, \dots, a_n\} \cap I = \emptyset$ implies $a_0 \in I$. Interpretation I is a *model* of a program P if each rule $r \in P$ is true under I. Finally, I is an answer set of P if I is a subset-minimal model of P^I, where P^I is defined as the program that results from P by deleting all rules that contain a default negated atom from I, and deleting all default negated atoms from the remaining rules.

Programs can yield no answer set, one answer set, or many answer sets. For instance, the program

$$\{p :- \text{not } q,\ q :- \text{not } p\} \tag{2}$$

has two answer sets: $\{p\}$ and $\{q\}$.

When we represent a problem in ASP, some rules "generate" answer sets corresponding to "possible solutions", and some "eliminate" the answer sets that do not correspond to solutions. The rules in program (2) are of the former kind; constraints are of the latter kind. For instance, adding the constraint $\perp :- p$ to a program P eliminates all answer sets of P containing p. In particular, adding $\perp :- p$ to program (2) eliminates the answer set $\{p\}$.

When we represent a problem in ASP, we often use special constructs of the form $l\{a_1, \dots, a_k\}u$ (called *cardinality expressions*) where each a_i is an atom and l and u are nonnegative integers denoting the *lower bound* and the *upper bound* of the cardinality expression [12]. Such an expression describes the subsets of the set $\{a_1, \dots, a_k\}$ whose cardinalities are at least l and at most u. In heads of rules, cardinality expressions generate answer sets containing subsets of $\{a_1, \dots, a_k\}$ whose cardinality is at least l and at most u. When used in constraints, they eliminate answer sets that contain such respective subsets.

A group of rules that follow a particular pattern can often be described in a compact way using *schematic variables*. For instance, we can write the program $p_i :- \text{not } p_{i+1}$, $(1 \le i \le 7)$ as follows:

$$\begin{array}{l} \mathit{index}(1),\ \mathit{index}(2), \dots, \mathit{index}(7), \\ p(i) :- \text{not } p(i+1), \mathit{index}(i). \end{array}$$

ASP solvers compute an answer set for a given program that contains variables after "grounding" the program, e.g., by the grounder `gringo` [13]. A grounder systematically replaces each rule r with variables by its ground instances that result from r by uniformly replacing each variable by constants from the program. Variables can also be used "locally" to describe a list of literals. For instance, the rule $1\{p_1, \dots, p_7\}1$ can be represented as $1\{p(i) : \mathit{index}(i)\}1$.

In addition to the constructs above, current state-of-the-art ASP solvers support many language extensions like *functions*, *built-in arithmetics*, *comparison predicates*, *aggregate atoms*, *maximisation* and *minimisation statements*, as well as *weak constraints*.

In the remainder of this paper, we use the syntax that is supported by the solver `clasp` along with the grounding tool `gringo` when presenting programs [14].

For illustrating problem solving in ASP, consider the following encoding of the 3-colorability problem (3COL):

```
colour(red;green;blue).
1{asgn(N,C):colour(C)}1 :- node(N).
:- edge(X,Y), asgn(X,C), asgn(Y,C).
```

The first rule abbreviates three facts that state that red, green, and blue are colours, respectively. The second rule is a choice rule. Its intuitive reading is that if `N` is a node, then both an upper bound and a lower bound on the number of colours assigned to this node, expressed by `asgn(N,C)`, is 1. This means that each node gets assigned precisely one colour from the set of available colours defined by `colour/1`. The last rule is a constraint that forbids that there is an edge between any two nodes with the same colour. If the above program is joined with facts over `edge/2` and `node/1` that represent a graph G, the answer sets correspond one-to-one to the valid 3-colourings of G.

Sometimes, one is not only interested in arbitrary solutions to a problem but in solutions that are optimal according to some preference relation. ASP solvers like `clasp` support optimisation statements that allow to express such preferences. For illustration, assume that, for some reason, we want to minimise the number of blue nodes in the above 3COL example. This can be expressed by simply adding the following minimise statement:

```
#minimize[asgn(N,blue):node(N)].
```

The meaning of such a statement is that `clasp` computes answer sets where the sum of literals `asgn(N,blue)`, where `N` is a node, is minimal among all answer sets.

III. SCA Computation

We now discuss how ASP can be used to generate SCAs. Our goal is not only to present approaches to compute generic SCAs, i.e., SCAs created without additional constraints or requirements, rather we want to demonstrate that ASP can be used as an efficient and effective declarative tool to compute SCAs tailored to specific test scenarios.

Ahead of our discussion in Section IV, addressing how different problem elaborations can be incorporated into a single answer-set program, we introduce an answer-set program for computing generic SCAs. We also introduce a new greedy approach that combines a simple variation of the basic ASP encoding with an iterative greedy procedure.

A. Basic Encoding

We first present an ASP program for computing (n, s, t)-SCAs with $t = 3$. We assume throughout that $s \ge 2$. Note that this program can be changed in a straightforward way

```
% guess sequence covering array
 sym(1..s). row(1..n).
 1{first(N,S):sym(S)}1 :- row(N).
1{next(N,S,T):sym(T)}1 :- first(N,S).
0{next(N,T,U):sym(U)}1 :- next(N,_,T).

% the happens-before relation
hb(N,X,Y) :- next(N,X,Y).
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).

% each symbol occurs once in each row
:- hb(N,S,S).
:- row(N), sym(S), first(N,T), S!=T,
   not hb(N,T,S).

% check if each 3-sequence is covered
threeSeq(X,Y,Z) :- sym(X;Y;Z),X!=Y,Y!=Z,X!=Z.
covered(X,Y,Z)  :- hb(N,X,Y), hb(N,Y,Z).
:- threeSeq(X,Y,Z), not covered(X,Y,Z).
```

Figure 1. ASP encoding $\Pi^3(n, s)$.

to obtain encodings for any fixed $t > 3$. An encoding for SCAs where t is not fixed can be obtained using *disjunctive ASP*—this is however beyond the scope of this paper.

1) Encoding: We start by expressing that the symbols of the array are integers between 1 and s, and row indices of the SCA correspond to integers 1 to n. Note that s and n function as parameters of the program:

```
sym(1..s). row(1..n).
```

For the representation of the SCA, we use the predicate next(N,X,Y) expressing that in row N symbol Y is the direct successor of X. We next state that in any row N (i) one symbol S occurs first, (ii) the first symbol S in row N has a direct successor T, and (iii) if T is consecutive to S, then there is at most one symbol U that is consecutive to T:

```
 1{first(N,S):sym(S)}1 :- row(N).
1{next(N,S,T):sym(T)}1 :- first(N,S).
0{next(N,T,U):sym(U)}1 :- next(N,_,T).
```

So far, the above conditions are only necessary conditions for an $(n, s, 3)$-SCA. We need further rules to guarantee that any row is a permutation of the symbols $\{1, \ldots, s\}$ and that coverage of all 3-sequences is achieved. We proceed by formalising the *happens-before* relation between two events. In particular, that one event symbol X occurs before another symbol Y in row N is represented by predicate hb(N,X,Y), which is simply the transitive closure of the next/3 relation:

```
hb(N,X,Y) :- next(N,X,Y).
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).
```

Directly expressing inductive definitions as above is a particular strength of ASP. Based on the happens-before relation, we can quite easily state that each event symbol has to occur precisely once in each row. We express this by means of two constraints. The reading of the first one is that it is forbidden that there is a row N such that a symbol S occurs before itself. The second constraint ensures that it is forbidden that there is a row N such that some symbol S different from the first symbol T does not occur after T. Together, the constraints imply that next/3 indeed represents permutations.

```
:- hb(N,S,S).
:- row(N), sym(S), first(N,T), S!=T,
   not hb(N,T,S).
```

It only remains to require that each 3-sequence of symbols is covered by some row. We use predicate threeSeq(X,Y,Z) to represent the 3-sequences that we want to cover. A 3-sequence is simply a 3-tuple of pairwise distinct symbols:

```
threeSeq(X,Y,Z) :- sym(X;Y;Z),X!=Y,Y!=Z,X!=Z.
```

A 3-sequence (X,Y,Z) is covered if X happens before Y and Y happens before Z in some row N. We finally define covered 3-sequences and forbid that a 3-sequence is not covered:

```
covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
:- threeSeq(X,Y,Z), not covered(X,Y,Z).
```

The entire ASP program $\Pi^3(n, s)$ with parameters n and s for generating $(n, s, 3)$-SCAs is given in Figure 1.

Intuitively, each answer set of program $\Pi^3(n, s)$ represents an $(n, s, 3)$-SCA. In fact, the answer sets of $\Pi^3(n, s)$ and the $(n, s, 3)$-SCAs are in a one-to-one correspondence. This relation can be formalised as follows:

Definition 2: An answer set X of $\Pi^3(n, s)$, for $s \geq 2$, *represents* an $n \times s$ matrix M iff for any i, $1 \leq i < s$, and any r, $1 \leq r \leq n$, $M_{r,i} = s_1$ and $M_{r,i+1} = s_2$ precisely in case X contains the atom $\mathtt{next}(r, s_1, s_2)$.

Proposition 1: Each answer set of $\Pi^3(n, s)$ represents a single $(n, s, 3)$-SCA, and each $(n, s, 3)$-SCA is represented by a single answer set of $\Pi^3(n, s)$.

For illustration, to compute a $(7, 5, 3)$-SCA, gringo and clasp can be invoked as follows:

```
gringo sca-3.gr -c n=7,s=5 | clasp.
```

File sca-3.gr contains program $\Pi^3(n, s)$. The gringo option -c n=7,s=5 instantiates the program parameters n and s to 7 and 5, respectively. Any resulting answer set corresponds to a $(7, 5, 3)$-SCA. For instance, in some answer set, the first row of the SCA M given in Section II-A is encoded by the atoms next(1,5,2),next(1,2,3), next(1,3,1),next(1,1,4). To compute more than one $(7, 5, 3)$-SCA, an upper bound on the number of answer sets that clasp should compute can be specified as an integer option (0 means that all answer sets are computed).

2) Discussion: Program $\Pi^3(n, s)$ nicely illustrates how challenging search problems can be concisely encoded using ASP: The program consists of only 12 rules that closely reflect the problem statement in natural language. We note that only little training time is needed to enable a tester to use ASP for test authoring. This is mainly because of the

Table I
UPPER BOUNDS FOR $\mathrm{SCAN}(s, 3)$ OBTAINED BY KUHN ET AL. AND OUR ASP ENCODING. A STAR INDICATES AN OPTIMAL BOUND.

s	n (Kuhn et al.)	n (ASP)
5	8	7*
6	10	8*
7	12	8*
8	12	8*
9	14	9
10	14	9
11	14	10
12	16	10
13	16	10
14	16	10
15	18	10
16	18	11
17	20	11

genuine declarative nature of ASP, which does not require specialised knowledge on data structures or algorithms. A more experienced ASP user needs about 15 minutes to develop a program such as the one given in Figure 1.

Also, by using our ASP encoding $\Pi^3(n, s)$ and the ASP solver `clasp`, we could improve known upper bounds for many SCAs significantly. A comparison of the SCAs generated using ASP and the greedy algorithm of Kuhn et al. is given in Table I. The SCAs that we have computed using ASP are publicly available [15]. Computation times for the reported upper bounds range from fractions of a second to about 20 minutes. We have considered strength 3 SCAs for 5 to 17 events. The known upper bounds reported by Kuhn et al. [3], [4] could be improved throughout. The more events are considered, the more drastic are the improvements; e.g., for 17 events, we need 45% less test sequences.

For small SCAs—viz. for 5 to 8 events—the new upper bounds are actually optimal bounds. Optimality of upper bounds was established using ASP itself. To show that an (n, s, t)-SCA is optimal, we try to compute an $(n-1, s, t)$-SCA. If this fails, i.e., the ASP solver terminates without returning an answer set, the (n, s, t)-SCA is indeed optimal. Since $\mathrm{SCAN}(8, 3)$=8, 8 is a trivial lower bound for any $\mathrm{SCAN}(s, 3)$ with $s > 8$. Note that greedy algorithms, or any approaches based on incomplete search, are unable to prove optimal bounds or to establish lower bounds at all.

A limitation of using the ASP encoding $\Pi^3(n, s)$ concerns scalability. Though memory usage is always limited by a polynomial with respect to the input parameters n and s, the runtime of `clasp` is worst-case exponential for encoding $\Pi^3(n, s)$. On the other hand, the greedy approach of Kuhn et al. seems to scale quite well; the authors report on SCAs of strength three and four for up to 80 events [4].

B. Greedy Algorithm

In the remainder of this section, we introduce and discuss an ASP-based greedy algorithm, inspired by that of Kuhn et al. [3], [4], for computing larger SCAs. The motivation to study such an algorithm is to combine the modelling capabilities of ASP, especially in the light of constraints and problem elaborations (as detailed in the next section), with the scalability of a greedy approach.

```
Require: $s$ is the number of symbols.
Ensure: $N$ represents an $(n, s, 3)$-SCA.
1: $N \Leftarrow \emptyset$
2: $n \Leftarrow 0$
3: repeat
4:     $n \Leftarrow n + 1$
5:     $X \Leftarrow$ answer set of $\Pi^3_{grdy}(s, n) \cup N$
6:     $N \Leftarrow N \cup X|_{\texttt{next/3}}$
7: until $N$ represents an $(n, s, 3)$-SCA
```

Figure 2. Greedy algorithm for computing an $(n, s, 3)$-SCA.

In this context, we also mention that the greedy algorithm of Kuhn et al. has a certain weakness, which is related to the heuristic that for any newly computed sequence the reverse sequence is added as well (cf. Section II). As we will show next, this makes the algorithm inherently unable to compute optimal SCAs in general. Actually, the inability to find optimal SCAs follows immediately from the observation that some optimal SCAs, e.g., $(7, 5, 3)$-SCAs, are of odd size. However, ASP can be used to show that even optimal SCAs of even size cannot be found by that greedy approach in general. The idea is to augment program $\Pi^3(n, s)$ by a rule that states that every second row is the inversion of the previous one. This is simply expressed by the following rule:

```
next(N,S,T):- row(N),next(N-1,T,S),N#mod2==0.
```

Here, predicate `#mod` is the usual modulo operation. Hence, the intuitive reading of this rule is that for any row with even index `N`, the `next` relation is the inverse of the `next` relation of the preceding row `N-1`. We know already from Table I that any $(8, 6, 3)$-SCA is optimal. However, $\Pi^3(8, 6)$ augmented by the above rule yields no answer set, which shows that $(8, 6, 3)$-SCAs cannot be computed by the greedy algorithm of Kuhn et al. [3], [4]. Next, we present an ASP-based greedy algorithm inspired by that of Kuhn et al. that does not rely on adding inverted rows.

1) Encoding: Figure 2 represents our ASP-based greedy algorithm for computing SCAs. The main idea is to compute one row of a SCA at a time instead of computing the entire array. In each iteration, one further row is computed using ASP where the number of covered 3-sequences is maximised. For this purpose, we use program $\Pi^3_{grdy}(s, n)$, which is depicted in Figure 3. Program $\Pi^3_{grdy}(s, n)$ takes the number s of events and a row index n as parameters. Both the ASP encoding and the greedy algorithm are introduced only for SCAs of strength 3. However, versions for computing SCAs of strength greater than 3 are obtained in a straightforward way. To obtain a program for strength 4 SCAs, for example, only the last two rules of $\Pi^3_{grdy}(s, n)$ have to be replaced by the following two rules:

```
covered(W,X,Y,Z) :- hb(n,W,X), hb(n,X,Y),
                    hb(n,Y,Z).
```

```
% guess single SCA row with index n
sym(1..s).
1 {first(n,S)  : sym(S)} 1.
1 {next(n,S,T) : sym(T)} 1 :- first(n,S).
0 {next(n,S,T) : sym(T)} 1 :- next(n,_,S).

% the happens-before relation
hb(N,X,Y) :- next(N,X,Y).
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).

% each symbol occurs once in each row
:- hb(S,S).
:- sym(S), first(n,T), S!=T, not hb(n,T,S).

% maximize coverage
covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
#maximize[covered(_,_,_)].
```

Figure 3. ASP encoding $\Pi^3_{grdy}(s, n)$.

```
#maximize[covered(_,_,_,_)].
```

Program $\Pi^3_{grdy}(s, n)$ is quite similar to $\Pi^3(n, s)$. However, each answer set of $\Pi^3_{grdy}(s, n)$ corresponds only to a single row with index n of an SCA. The idea is to represent preceding rows with index 1 to $n - 1$ by means of facts `next/3`. These facts are joined with $\Pi^3_{grdy}(s, n)$. Then, the answer sets of $\Pi^3_{grdy}(s, n)$ correspond to those rows that obtain maximal coverage of previously uncovered 3-sequences. The encoding follows the *guess, check, and optimise pattern*, hence we use guessing rules to span the search space, constraints to filter unwanted solution candidates, and rules that express a preference relation on answer sets. In particular, rule

```
#maximize[covered(_,_,_)].
```

states that we seek for answer sets with a maximal number of covered 3-sequences.

The algorithm itself is rather simple, see Figure 2: It takes parameter s as input and computes an $(n, s, 3)$-SCA. Initially, the set N that represents a (partial) SCA by means of facts `next/3` equals the empty set. In each iteration, $\Pi^3_{grdy}(s, n) \cup N$ are used to compute the next row of the SCA that obtains maximal increase of previously uncovered 3-sequences. The respective `next/2` facts for that row are then added to N. This procedure iterates until no uncovered 3-sequences are left (the ASP solver itself will indicate that no further optimisation is possible). Since the computation of optimal answer sets can become very time consuming, we additionally impose an upper bound on the time that is spent for optimising answer sets, thus improvements in each step will not be maximal in general. However, this seems to be a reasonable compromise regarding runtime and the size of computed SCAs. The time limit for computing a single row ranged from 10 seconds to several minutes, depending on the problem size.

Table II
COMPARISON OF OUR GREEDY ASP APPROACH AND THAT OF KUHN ET AL. [3], [4]: UPPER BOUNDS FOR SCAN$(s, 3)$ AND SCAN$(s, 4)$.

s	$t = 3$		$t = 4$	
	Kuhn et al.	ASP	Kuhn et al.	ASP
10	14	11	72	55
20	22	19	134	104
30	26	23	166	149
40	32	27	198	181
50	34	31	214	-
60	38	34	238	-
70	40	36	250	-
80	42	38	264	-

2) Discussion: Table II summarises a comparison of our greedy ASP algorithm with the greedy algorithm of Kuhn et al. [3], [4] for strength 3 and 4 SCAs involving 10 to 80 events. For strength 3 SCAs, our algorithm is competitive with that of Kuhn et al. and upper bounds could be improved throughout by some rows. For strength 4 SCAs, the greedy ASP approach is feasible for up to 40 symbols where upper bounds could be improved even more drastically than for strength 3 SCAs. However, we were not able to compute SCAs for 40 to 80 symbols, which shows a limitation of our ASP-based approach that is probably acceptable unless the need for larger instances with a high level of interaction is indeed motivated by some application scenario. This limitation basically comes from the huge number of 4-sequences that need to be covered and that are represented by the program. Here, it is to mention that scalability is certainly a characteristic strength of the simple greedy algorithm of Kuhn et al., since dedicated data structures, e.g., efficient bit-vectors, can be used for representing covered sequences. However, by using ASP we get better bounds for 3-SCAs for up to 80 symbols and can also improve bounds for 4-SCAs for up to 40 symbols. Again, we emphasise that our goal is not to compute generic SCAs but to allow a tester to express different requirements with little effort, by adding or changing some rules of the ASP program, which can readily be done using the greedy ASP approach. We pursue this issue in the next section.

IV. PROBLEM ELABORATIONS

Next, we turn to the actual strengths of using ASP as an elaboration tolerant representation formalism for event sequence testing. We describe how ASP can be used for generating SCAs in a scenario that involves additional constraints and other problem variations that make it impossible to directly use precomputed SCAs. In particular, we use a *real-world testing problem* described by Kuhn et al. [3], [4] for making our point. The specification of this testing problem is as follows: There are 5 different devices that have to be connected to a laptop. These devices can be connected before or after a boot-up phase. Further actions that have to be performed on the laptop are opening an application and initiating a scanning process. The peripherals can be connected to the laptop in any order; however, the order of events influences the functionality of the system. Thus, SCAs lend themselves as a basis for a suitable testing plan.

There are 8 events relevant for testing: connecting devices (p1,...,p5), booting the system (boot), starting an application (appl), and running a scan (scan). Testing in this scenario is rather time consuming since it requires setting up the system manually. Therefore, obtaining an optimal test plan is a clear desideratum. Following Kuhn et al., only SCAs of strength 3 are considered to keep the size of the test plan reasonable.

A. Forbidden Sequences

For 8 events, optimal SCAs of strength 3 comprise 8 rows. However, we cannot use precomputed $(8, 8, 3)$-SCAs since certain constraints regarding the order of events have to be taken into account. While most events can happen in any order, starting the application cannot happen before the system is booted, and running a scan requires that the application is already running.

1) Encoding: Instead of covering all 3-sequences, we want to generate SCAs such that (i) in each row, boot happens before appl and appl happens before scan, and (ii) all 3-sequences such that boot happens before appl and appl happens before scan are covered by at least one row. We only have to slightly modify program $\Pi^3(n, s)$ to account for (i) and (ii). First, instead of integers to denote events, we would like to use more descriptive constant symbols. Thus, we replace sym(1..s) in $\Pi^3(n, s)$ by

```
sym(boot; p1; p2; p3; p4; p5; appl; scan).
```

Concerning (i), we define which orderings are excluded and add a respective constraint that forbids that event a happens before b if "a before b" is excluded.

```
excluded(scan,appl).
excluded(appl,boot).
excluded(X,Z) :- excluded(X,Y),excluded(Y,Z).
:- hb(_,X,Y), excluded(X,Y).
```

Regarding (ii), we simply define those 3-sequences that are not consistent with the excluded orderings as already covered:

```
covered(X,Y,Z) :- excluded(X,Y), sym(X;Y;Z).
covered(X,Y,Z) :- excluded(X,Z), sym(X;Y;Z).
covered(X,Y,Z) :- excluded(Y,Z), sym(X;Y;Z).
```

We denote the resulting program as $\Pi^3_1(n)$.

2) Discussion: Recall that for 8 symbols, $(8, 8, 3)$-SCAs are optimal. Since, $\Pi^3_1(8)$ does not yield any answer set, it follows that the stipulation on admissible orderings requires additional rows. In this case, this is because the number of 3-sequences that can be covered by a single row is reduced if certain events are required to happen in a strict order. Indeed, a solution for $\Pi^3_1(9)$ can be computed, hence 9 is an optimal bound for an SCA satisfying that each row is consistent with the specified ordering constraints. The solver clasp needs fractions of a second to find an SCA of size 9 and about one minute for checking optimality.

B. Redundant Sequences

Besides forbidden orderings, we also have to deal with redundant sequences: If devices are connected to the laptop before the boot-up phase, the order is not relevant. In fact, we only require strength 3 coverage for events p1,...,p5, appl, and scan. Concerning the interaction of events p1,...,p5, and boot, we regard strength 2 coverage as sufficient, i.e., we are only interested in whether the connection of the peripherals happens before or after the boot-up phase. Hence, we need a variable strength SCA, in which we seek to have strength 2 coverage for one set of events and strength 3 coverage for another one.

1) Encoding: First, we add two sets of facts to declare the sets of events for which we want to obtain strength 2 and strength 3 coverage, respectively:

```
threeWay(p1; p2; p3; p4; p5; appl; scan).
twoWay(boot; p1; p2; p3; p4; p5).
```

Next, we have to modify some rules where appropriate. In particular, we only want to cover 3-sequences over symbols from threeWay/1. Hence, we rewrite rule

```
threeSeq(X,Y,Z) :- sym(X;Y;Z),X!=Y,Y!=Z,X!=Z.
```

into

```
threeSeq(X,Y,Z) :- threeWay(X;Y;Z),
                   X!=Y, Y!=Z, X!=Z.
```

To address 2-way coverage of the symbols from twoWay/1, we add two further rules:

```
covered(X,Y) :- hb(_,X,Y).
:- twoWay(X;Y), X != Y, not covered(X,Y).
```

The resulting program is denoted by $\Pi^3_2(n)$.

2) Discussion: Program $\Pi^3_2(n)$ incorporates both forbidden configurations and redundant sequences. Respective SCAs can be obtained for $n = 8$ already. SCAs of size 8 are indeed optimal arrays, which follows from the observation that $\Pi^3_2(7)$ yields no answer set at all. It takes on average 0.1 seconds to compute the first answer set of a size 8 SCA when using clasp as ASP solver. Showing optimality, i.e., that no size 7 SCA exists, needs several minutes.

The solution approach of Kuhn et al. uses a pre-computed $(12, 7, 3)$-SCA to account for the seven events p1,...,p5, scan, and appl. In a post-processing step, rows that are not consistent with the ordering constraints (cf. Section IV-A) are replaced. However, this requires that further rows are added to preserve coverage. Then, in a further manual post-processing step, to account for the two-way coverage with respect to events p1,...,p5, and boot, Kuhn et al. add boot as the first event of each row. Finally, an additional row is added, in which all events p1,...,p5 are arranged prior to boot, thereby obtaining strength 2 coverage between

Table III
TEST PLAN OF SIZE 8 FOR THE LAPTOP APPLICATION OBTAINED FROM AN ANSWER SET OF $\Pi^3_4(8)$.

row	event 1	event 2	event 3	event 4	event 5	event 6	event 7	event 8
1	p3(l)	p2(r)	p1(b)	p4	boot	appl	scan	p5
2	boot	p4	p1(r)	appl	p5	p3(l)	scan	p2(b)
3	boot	appl	scan	p1(r)	p2(b)	p4	p3(l)	p5
4	p1(r)	p2(b)	p5	p3(l)	boot	appl	scan	p4
5	boot	p3(b)	p5	p1(r)	appl	p4	p2(l)	scan
6	p4	boot	p2(b)	p5	appl	p1(l)	scan	p3(r)
7	boot	appl	scan	p5	p3(l)	p4	p2(b)	p1(r)
8	p5	boot	p2(l)	p4	p3(r)	appl	scan	p1(b)

boot and events p1,...,p5. The resulting array consists of 18 rows.

The first thing to note is that using ASP enabled us to easily embed the additional requirements directly in the ASP program rather than employing an ad hoc and mostly manual approach. Furthermore, using ASP significantly reduced the size of the resulting SCA by 55.56% (cf. Table III).

C. Adding Attributes to Events

The next problem elaboration that we consider is related to the way the peripherals are connected to the laptop. Devices p1, p2, and p3 have to be connected to USB ports. Three ports are available: left, right, and back. In each test sequence, one port has to be assigned to a USB device.

1) Encoding: Predicate port(N,X,Y) states that USB device X is connected to port Y in row N of the array. This assignment should satisfy the following coverage criteria: (i) each USB device has to be connected to each port at least once and (ii) connections to the ports after the boot event should be made in any possible order. The above requirements can be formalised using few further rules.

In the following rules, we first specify the USB ports and devices. Then, it is expressed that each USB device is assigned to precisely one port in each test sequence. Finally, USB devices must not be connected to the same port in any sequence.

```
usbPort(right; left; back).
usbDevice(p1; p2; p3).
1{port(N,X,Y):usbPort(Y)}1 :- row(N),
                              usbDevice(X).
:- port(N,X,Y), port(N,Z,Y), X != Z.
```

Next, we state coverage criterion (i):

```
portCov(X,Y) :- port(N,X,Y).
:- usbDevice(X),usbPort(Y),not portCov(X,Y).
```

Lastly, we add rules for coverage criterion (ii):

```
portSeq(X,Y,Z) :- usbPort(X;Y;Z),
                  X!=Y,X!=Z,Y!=Z.
seqCov(N,X,Y,Z):-hb(N,boot,X),hb(N,X,Y),
                 hb(N,Y,Z).
pSeqCov(R,S,T) :- seqCov(N,X,Y,Z),
      port(N,X,R), port(N,Y,S), port(N,Z,T).
:- portSeq(X,Y,Z), not pSeqCov(X,Y,Z).
```

Let us denote the resulting program by $\Pi^3_3(n)$.

2) Discussion: Note that the additional conditions regarding the USB ports do not result in larger SCAs, still SCAs of size 8 can be obtained by computing the answer sets of $\Pi^3_3(8)$. Clearly, 8 is also an optimal bound. The runtime of the ASP solver is not affected by the additional requirements.

Kuhn et al. deal with the issue of USB ports by adding respective port assignments in a post-processing step once an SCA is computed. However, they do not provide details on which basis this is done, i.e., it is not clear if or in what sense they strive for systematic coverage.

D. Expressing Preferences

Any answer set of $\Pi^3_3(n)$ represents one admissible test plan for the application under test. Although each such SCA satisfies all of the requirements discussed so far, different SCAs could differ in their fault detection potential.

We next augment program $\Pi^3_3(n)$ by rules that state a preference relation among solutions, similar to program $\Pi^3_{grdy}(\cdot,\cdot)$ from the previous section. In particular, although any SCA guarantees full 3-way interaction coverage for some specified events, the degree of 4-way coverage of events may differ from one SCA to another. We will use the number of covered 4-sequences as discrimination criterion regarding the quality of solutions and consequently prefer SCAs that cover more 4-sequences over SCAs that cover fewer.

1) Encoding: We define program $\Pi^3_4(n)$ as $\Pi^3_3(n)$ augmented by the following rules:

```
covered(W,X,Y,Z) :- hb(N,W,X),hb(N,X,Y),
                    hb(N,Y,Z).
#maximize[covered(_,_,_,_)].
```

The first rule defines which 4-sequences are covered, the second rule states that the number of covered 4-sequences should be maximised.

2) Discussion: An SCA of size 8 corresponding to an answer set of $\Pi^3_4(8)$ is given in Table III. In the computation of the SCA, clasp has been configured to optimise a solution until no improvements can be found for 15 minutes.

On the other hand, Kuhn et al. has not handled preferences over solutions at all. The algorithm of Kuhn et al. is tailored for computing a single SCA. Thus, it may be hard to use such an algorithm to directly deal with optimisation issues, since this requires that solutions should be efficiently enumerated.

This case study demonstrates that often generic SCAs cannot be used in a real world scenario without significant modifications. In general, such modifications lead to a considerable overhead or are not feasible at all. By using ASP, however, a test author has a tool to state different requirements relevant for individual scenarios. Often, this will need only little effort such as adding few rules.

V. Related Work

Since the approach of Kuhn et al. [3], [4] is based on a greedy algorithm for generating SCAs, which have to be modified in a post-processing step to meet different user requirements, the ASP-based approach introduced in this paper is the first account of an approach for directly generating SCAs in the presence of expressible constraints and problem elaborations.

Closely related to our work are techniques for computing covering arrays (CAs), which we will review next. There, greedy algorithms that construct one row at a time are quite common. The most prominent representative is the AETG system [16]. Our greedy approach to compute SCAs is close in spirit to AETG-like algorithms since it also proceeds row by row. Also, meta-heuristics, like simulated annealing, tabu search, or genetic algorithms, have been applied for constructing CAs [17], [18], cf. respective overview articles [1], [2]. However, neither greedy techniques nor meta-heuristics can guarantee optimal bounds.

As a complete method being able to establish optimality of arrays, different SAT encodings have been considered [19], [20]. A distinctive feature of ASP compared to SAT is the high-level modelling capabilities of ASP that allow to model problems concisely at the first-order level as demonstrated by our SCA encodings. SAT is certainly a promising approach for tackling problems described in Section III, i.e., for computing SCAs and checking optimality of upper bounds. However, the problem variations discussed in Section IV require a formalism that allows for elaboration-tolerant representations, which is not a characteristic feature of SAT. Regarding modelling, it is to mention that Hnich et al. [19] and Banbara et al. [20] initially considered constrained programming (CP) models, which are subsequently translated to SAT. Though this has not been considered, further constraints, at least forbidden tuples, could be incorporated rather easily into the CP model. A comparison of ASP and constrained (logic) programming (CLP) is given in a related article [21]. There, the authors conclude that ASP allows for more declarative and concise problem representation and is easier to learn for newcomers than CLP. We also mention in passing that a vital aspect of the CP models was related to breaking symmetries, which obscures problem representation somewhat. Though symmetry breaking is also an issue in ASP, we experienced that adding symmetry-breaking constraints to our ASP programs has a quite negative effect on the performance of ASP solvers for improving upper bounds.

Cohen, Dwyer, and Shi [22], [23] introduced approaches that integrate techniques for generating covering arrays with SAT to deal with constraints. Forbidden tuples are represented as Boolean formulas and a SAT solver is used to compute models. They integrated SAT with greedy AETG-style algorithms and also with simulated annealing. Hence, their approach is closely related to our integration of ASP into a greedy procedure. Calvagna and Gargantini [24] follow a similar approach but they use an SMT solver instead of a SAT solver, which offers a richer language than plain SAT solvers. In their approach, constraints are stated as formal predicate expressions. Besides SMT, Calvagna and Gargantini also considered a model checker for verifying test predicates.

Bryce and Colbourn [25] distinguish forbidden tuples and tuples that should be avoided. They refer to the latter as soft constraints and they present an algorithm for generating CAs that avoids the violation of soft constraints. However, their algorithm cannot guarantee that certain tuples are avoided, hence it cannot deal with forbidden tuples or other hard constraints. Using ASP, soft constraints can be easily expressed by means of minimise or maximise statements. We illustrated in the previous section how one can combine hard integrity constraints with soft constraints to express that uncovered 4-sequences should be avoided.

VI. Conclusion and Future Work

In this paper, we dealt with the generation of SCAs, which have recently been advocated as suitable combinatorial design for event sequence testing [3], [4]. In particular, we applied ASP as a declarative approach for generating SCAs. While the only previously introduced algorithm is an AETG-like greedy algorithm [3], [4], ASP can be used as an exact method that combines high-level modelling capabilities with highly performative search engines [8].

To summarise, our contribution is two-fold: On the one hand, we introduced and showed feasibility of a new approach for generating SCAs that can be readily used as it is. On the other hand, we regard this work as a contribution towards methodology. While ASP is well established in other communities as a method to address problems from the area of artificial intelligence and knowledge representation, too little is known about ASP in the software-engineering community. Hence, we want to promote ASP as an approach to tackle challenging problems in the realm of combinatorial testing. Besides improving the state-of-the-art of event sequence testing, our aim is to show that ASP provides a tool that enables a tester to rapidly specify problems and to experiment with different formulations at a purely declarative level. ASP solvers are then used for computing solutions without the need of post-processing steps or developing dedicated algorithms.

For future work, we plan to deal with versions of SCAs for different testing applications like testing of concurrent programs where the order of shared variable accesses was identified as crucial for triggering certain bugs that are otherwise hard to evoke [6], [26].

Acknowledgment

This work was supported by the Austrian Science Fund (FWF) under project P21698. We also would like to thank D. Richard Kuhn for providing us with related work [4].

References

[1] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," *Software Testing, Verification & Reliability*, vol. 15, no. 3, pp. 167–199, 2005.

[2] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv*, vol. 43, no. 2, pp. 11:1–11:29, 2011.

[3] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," NIST National Institute of Standards and Technology, NIST Special Publication 800–142, October 2010.

[4] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," 2010, submitted for publication. Available at http://csrc.nist.gov/groups/SNS/acts/documents/event-seq101008.pdf.

[5] M. J. Harrold and B. A. Malloy, "Data flow testing of parallelized code," in *Proceedings of the 8th International Conference on Software Maintenance (ICSM 1992)*. IEEE Computer Society Press, 1992, pp. 272–281.

[6] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2007, pp. 533–536.

[7] C. Baral, *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, 2003.

[8] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński, "The second answer set programming competition," in *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, ser. LNCS, vol. 5753. Springer, 2009, pp. 637–654.

[9] "Combinatorial testing for event sequences," http://csrc.nist.gov/groups/SNS/acts/sequence_cov_arrays.html, last visited: July 18, 2011.

[10] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set solving," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*. AAAI Press/MIT Press, 2007, pp. 386–392.

[11] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *Proceedings of the 5th Logic Programming Symposium*, MIT Press, 1988, pp. 1070–1080.

[12] P. Simons, I. Niemelä, and T. Soininen, "Extending and implementing the stable model semantics," *Artificial Intelligence*, vol. 138, no. 1–2, pp. 181–234, 2002.

[13] M. Gebser, T. Schaub, and S. Thiele, "Gringo: A new grounder for answer set programming," in *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, ser. LNCS, vol. 4483. Springer, 2007, pp. 266–271.

[14] "Potassco—the potsdam answer set solving collection," http://potassco.sourceforge.net, last visited: July 18, 2011.

[15] http://www.kr.tuwien.ac.at/research/projects/mmdasp/collection-of-scas.tar.gz, last visited: July 18, 2011.

[16] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Trans. Software Eng.*, vol. 23, no. 7, pp. 437–444, 1997.

[17] M. B. Cohen, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, 2003, pp. 38–48.

[18] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, no. 1-2, pp. 143–152, 2004.

[19] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, no. 2-3, pp. 199–219, 2006.

[20] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue, "Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers," in *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*, ser. LNCS, vol. 6397. Springer, 2010, pp. 112–126.

[21] A. Dovier, A. Formisano, and E. Pontelli, "An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems," *J. Exp. Theor. Artif. Intell.*, vol. 21, no. 2, pp. 79–121, 2009.

[22] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 16th ACM/SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2007, pp. 129–139.

[23] ——, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Software Eng.*, vol. 34, no. 5, pp. 633–650, 2008.

[24] A. Calvagna and A. Gargantini, "A formal logic approach to constrained combinatorial testing," *Journal of Automated Reasoning*, vol. 45, no. 4, pp. 331–358, 2010.

[25] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information & Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[26] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2008, pp. 329–339.

A Test Case Suite Generation Framework of Scenario Testing

Ting Li [1,3]
1) Shanghai Development Center of Computer Software Technology
Shanghai, China
lt@ssc.stn.sh.cn

Zhenyu Liu [2]
2) Shanghai Key Laboratory of Computer Software Testing and Evaluating
Shanghai, China
{lzy, jiangx}@ssc.stn.sh.cn

Xu Jiang [2]
3) Shanghai Software Industry Association
Shanghai, China
lt@softline.sh.cn

***Abstract*—This paper studies the software scenario testing, which is commonly used in black-box testing. In the paper, the workflow model based on task-driven, which is very common in scenario testing, is analyzed. According to test business model in scenario testing, the model is designed to corresponding test case suite. The test case suite that conforms to the scenario test can be obtained through test case generation and test item design. In the last part of the paper, framework of test case suite design is given to illustrate the effectiveness of the method.**

Keywords-test case; software test; scenario testing; test suite.

I. Introduction

Software testing is the main activity of software quality. The goal of software testing is to validate whether software is good or conform to the initial requirement. However, software engineers always consider that software testing should be terminated under certain conditions. The adequacy of software testing is an important factor according to testing purpose. It is generally agreed that when software testing reaches expected test purpose, the software testing activities could be terminated. Thus, the quality and overhead cost of software testing can be considered fully and controlled effectively.

Nowadays, business processes become complicated by the development of technology and information. At present, many factors led to the e-business more complicated, such as business logical become complexity, component-based development widely accepted and complete workflow processes scattered in various business components. The component-based software led to process the data flow and control flow the more tightly and more complicated. Therefore, the scenario testing and verification has become important increasingly before system runtime. The scenario testing can be regarded as an independent test, which becomes an important part of black box testing. On the one hand, many business processes adopted e-business management. The traditional paper-based business model was replaced. On the other hand, uncertainty of system requirement brings risks to scenario testing. So, it is necessary to consider scenario testing gradually.

Software testing is the critical activity in the software engineering. However, some research shows the design of test cases will cost much time during software testing. While many business workflow management systems have emerged in recent years, few of them provide any consideration for business workflow verification.

The research work on business scenario testing and validation is less. As for the actual development of the application system, the every operation in business is designed and developed well. However, business processes testing and requirements verification are very important and necessary for software quality. The scenario testing satisfies software requirement through the test design and test execution.

Scenario testing is to judge software logic correction according to data behavior in finite test data and is to analyze results with all the possible input test data. It is generally considered that test design can evaluate software testing quality and select the test data. Scenario-based testing focuses on what the end-user does, not what system does. The flow path and boundary conditions are used to design test data corresponding to business scenario. Therefore, the main purpose of scenario testing is to find business flow interaction defects as possible. The test results also help to record results of software runtime during test execution.

Based on the existing test purpose of scenario-testing, in this article, we'll focus our research efforts on test case design of scenario testing. In the first section, we introduce the business scenario testing and its importance. And then, in the second section, we give out a business scenario model. In the third part, a test case suite model is proposed. The generation method of design test case is introduced in the fourth section, according to the requirements of business scenario which conforms to test case suite model. The fifth part gives related works and the conclusion is drawn with a corresponding discussion in the final section.

II. Business Scenario Model

Firstly, we give the typical business scenario model, which supports business model. The business model consists of three elements: business workflow, business scenario and basic operation.

A. Concepts

The business workflow indicates the typical business to accomplish the basic business process. Indeed, the basic workflow always consists of different scenario in the business, and any scenario corresponding to the business workflow. Therefore, we give the definition of the business scenario model.

Firstly, notations are introduced (see Table 1).

TABLE I. NOTATIONS

Symbol	Implication
BW	Business Workflow
SC	Scenario
OP	Operation
I	Input Set
O	Output Set
D	Data, Test Data
U	User
R	Role
S	State
PS	Previous State
SS	Success State
TR	Test Result
TC	Test Case
TCS	Test Case Suite

Definition 1: (Business Workflow) Assumes the business workflow, there are some business models for the fully business operations.

BW= {I, O, SC, D}, in BW, I am input set, O is output set, SC is related the scenarios, D is the test data corresponding to specific scenarios.

Definition 2: (Scenario) As for business process, there are scenarios in the business workflow. Every scenario could represent the possible workflow path in business flow.

SC= {I, O, OP, R, U, D, S, RE}, in SC, I and O indicate input and output separately, OP is the operation related to the current scenario, U and R are abbreviated of user and role separately. User and Role are the execute member of the operation. S is the state which describes the workflow.

Definition 3: (Operation) In any scenario of business workflow, operation is the basic element which accomplishes the specific function or function collection.

OP= {PS, SS, UR, D}, in the OP, PS and SS are previous state and successive state, which indicate the workflow state before and after the operation.

Definition 4: (Role) There are three kinds of roles, one is the workflow initiator (Sender), other two are workflow receiver and workflow informed person the workflow-based business model.

Definition 5: (User) Assumes the business process, there are multiple operators, operator is a member of role for specific operation.

For the two operations are carried out by the two users in workflow, there are seven typical models for operation. In these seven models, one model is concurrency or parallel, and the other models are different orders of successive operations. For the two operations and two users, two operations implement the following executable sequence (see Table 2).

Now, many collaborative business software systems are followed these two models: concurrently and end-start. Other models would exist in some special collaborative software system.

TABLE II. MODELS OF TWO OPERATIONS

Model	Description	Notation
Concurrent	Starts at the same time	A||B
Switch	Any one starts	A|B
Start Start	One start and another state	ss(A, B)
End Start	One finish then another state	es(A, B)
End End	One finish then another could finish	ee(A, B)
Start End	One start the another could finish	se(A, B)
Loop	Repeat when finish	[A]

Based on the workflow definition, four typical business models: sequence, parallel, switch and loop.

B. Model

- Sequence structure is used to define some activities in sequence execute order. In Fig. 1, where OPa, OPb are two independent tasks, OPb is defined as S2 and the causal state is S3. The sequence model could be denoted as es(OPa,OPb).

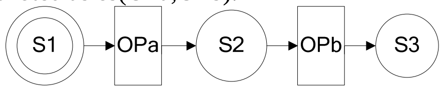


Figure 1. Sequence Structure

- Parallel structure used to define the order is not strictly executable. Every task in branch is not running at the same time, it needs to use two basic workflows: 'branch' and 'connection'. Fig. 2, where OPa, after implementation by S1 directly transferred to the S2 and S4, then the two tasks OPb, OPc can be executed separately, therefore is a parallel relationship, and then performed the OPb, OPc, after its implementation by the OPd. Fig. 2 could be denoted as es(OPa, (OPb || OPc), OPd).

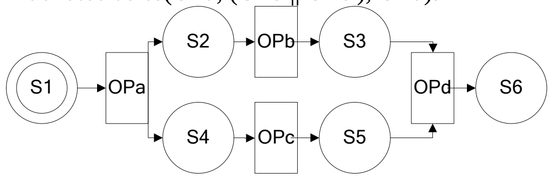


Figure 2. Parallel Structure

- Switch Structure is similar to the parallel structure, but the condition is selected according to state of S2/S3, rather than parallel structures, which OPb and OPc is performed at the same time. This model could be denoted as es(OPa, (OPb|OPc), OPd).

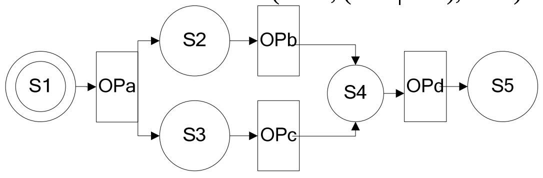


Figure 3. Branch/Select Structure

- Loop Structure. This structure is used to define that repeat the implement is needed in many tasks. The loop structure shown in Fig. 4 is modeled as es(OPa, [es(OPb,OPc)]). Here, operation OPb is executed repeatedly in loop structure.

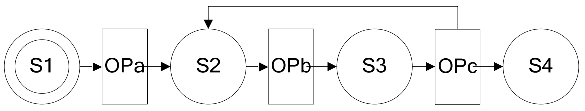


Figure 4. Loop Structure

III. Test Case Suite Model

During software testing, a standard test script template is provided to facilitate execute test case and collect test results. The standard template not only is the standardized collection process of quality elements, but also can assist comparative analysis of different test results.

The test script is the basic element for test execution. Therefore, test script design is accounting for much time on test design. It is necessary to find generic test cases in order to reduce the cost of the test case design for the further test case design and test execution. For some same operation in business, the reuse test script can accelerate the test case design. The process of automatically generating test cases can also improve the efficiency of test design [1]. As for testing activities is high cost work, test design is accounting for the workload and ability of test designer, the test case reuse method is adopted to improve the design efficiency. The reuse test case can reduce the design costs if used existing test case which have been used. The test case reuse technique can improve the test efficiency and reduce test cost [2].

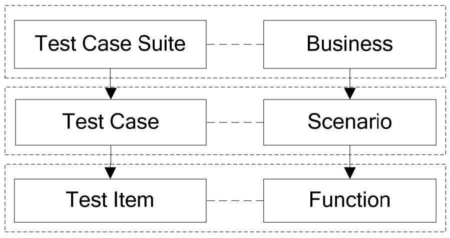


Figure 5. Test Case Suite Level Model

The test case suite is top level in the test suite model. The test case suite, also named test suite, is the collection of test case. The test case suite is corresponding to the business model. The test case is the middle level in Fig. 5. The every test case in test case suite is corresponding to the specific scenario in the business model. In other words, the test case could accomplish one of the business flows. The business flow always consists of many possible workflow paths. For example, as for the bank transaction workflow, the same bank transaction in different bank could design two scenarios in the transaction business workflow. The example demonstrates the relationship between the business and scenario. There are more scenarios than the examples in the actual test requirement. Therefore, the different methods of transaction could design for different test case in the test suite of business.

The test case is related to specific scenario in business workflow. In the business scenario, many tasks constitute the integrated scenario. The operation is executed by user with authority role. The relation between two operations in the specific scenario should be modeled, which introduced in section 2. The operations could be converted into test items in test suite model. In the test suite model, the operation is the basic element. During software testing, test item is the basic elements during testing execution.

Definition 6: (Test Case Suite) Test case suite consists of test purpose and related test cases. Test case consists of related scenario, test method, expected test result and test item collection. BNF is shown as Table 3.

TABLE III. BNF Of Test Case Suite

```
<TestCaseSuite>::=<BusinessRule><TestPurpose><User><Role>
            <TestType><Data>{<TestCase>}
<BusinessRule>::== /*refer to the business model */
<TestPurpose>::= /* test goal for the business*/
<TestType>::= function | performance | security | others
<User>::=/*  user info for the operator */
<Role>::={<User>} /* usergroup */
<TestCase>::=<Scenario><Method>{<TestItem>}<TestResult>
< Scenario >::= /* test case state and its function when start */
<Method>::= manual | automated
<TestResult>::=  /* the expected test result for test case */
<TestItem>::=<TestInput><TestOutput><TestData><TestOracle>
<TestInput>::=   /* operation procedure and input information*/
<TestData>::= /* test data collection refers to the input */
<TestOracle>::= /*expected the result based on input and data*/
<TestOutput>::= /* software output information*/
```

Definition 7: (Test item) Test item is an individual test step in the test case model, including the test input, test data, test oracle and test output.

Test case consists of many test items. The test items belong to the relevant test case. The test item is the part of the test case and every test item is run during the test execution. The different test scenario can add different test items to fulfill the test case. Therefore, scenario testing can be reconstructed according to the description of the test case. Test item is a fundamental element to construct test case.

IV. Generation Method

The test case suite should be designed according to the business model. The business model and workflow is complicated in that the possible business flow path should be considered completed for the validation and verification.

A. Generation Framework

The paper gives an integrated framework. Fig. 6 gives the test case suite generation framework. The framework helps test designer to design test case suite, test case and test item. The business model and test purpose are input items. The important activities in test design are analyzing path and related operation. The analyze path is to get the possible path according to the business model. The all possible paths are to help generate related test case. The purpose of analyzing operation is to design the test item. The test case, test item, and their test data constitute the test case suite, which is needed for test execution.

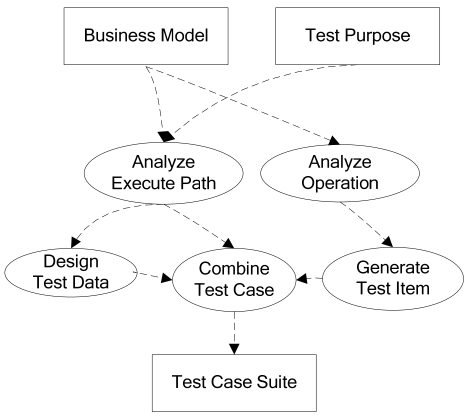


Figure 6. The framework of test case suite generation

- Analyze Execute Path

The test scenarios need to be analyzed all possible execution paths. According to the model presented earlier and test purpose, the designer should analyze the paths to accomplish the specific business. Test cases need to be considered that the coverage of business processes should conform to test purpose. The typical logic coverage consists of statement coverage, decision coverage, condition coverage, decision/condition coverage.

Test requirements consist of various logical coverage, such as path coverage, decision coverage, condition coverage and condition/decision coverage, etc. The different coverage requirements impact on test data design. The decision coverage is to determine the true and false values in one statement.

Here, we consider the situation, including two conditions, the one condition (C1) is x<0 or y>5 and the other condition (C2) is x>2 and y>3. The x value and y value constitute collection 1 and collection 2, the detail value and condition as below. The collection 2 satisfies the condition/decision coverage.

Collection 1(C1)

x	y	x<0	y>5	x>2	y>3	C1	C2
-1	3	T	F	F	F	T	F
2	6	F	T	F	T	T	F
3	2	F	F	T	F	T	T

Collection 2(C2)

x	y	x<0	y>5	x>2	y>3	C1	C2
-1	3	T	F	F	F	T	F
2	4	F	F	F	T	T	F
3	2	F	F	T	F	T	T

The data size should be further considered. The more data size, the more test execution time and cost. That is to say, the minimum number of test cases improved test efficiency. The design test data will affect the size of test case. Although different test data could achieve the same test purpose eventually, the less test data will reduce the test execution time.

It can be seen, the value y in condition 2 is not meet is true of the condition y> 5, so a new value is needed to design. If consider further optimization, the collection can be two data sets, which reduce one data set compare to original data sets. The final collection is:

x	y	x<0	y>5	x>2	y>3	C1	C2
-1	3	T	F	F	F	T	F
3	4	F	T	T	T	F	T

- Analyze Operation

Operation analysis will get the basic operation based on business model, including test items and test cases. Each test item is corresponding to a basic operation. The basic operations maintain consistency with the test data.

- Combine Test Case

Each operation will be designed to one test item. Test case consists of test procedures and related test data. Test procedures are sequence collection of test items, which sequence corresponds to execution path that analyzed during test design. The execution path is transformed into a series of branches that determine the condition and coverage. The user role information is also to be considered.

Through a set of test cases, test designer can be combined into a test case suite based on business models and their test purpose.

B. Algorithm

Here, we develop the algorithm for generating the test case suite, test cases and related test data. The algorithm analyzes the business rule and transverse all possible workflow paths. The every execution path is corresponding to the test case. The all test cases are composed into an integrated test case suite, that is to say, the test case suite consists of all possible execution paths. Indeed, some execution paths may be invalid due to the contradiction between test data and business logical. The some execution paths will be reduced according to test data.

```
INPUT: SC,OP,TR
OUTPUT: TCS
Begin
  OP'= {};
  EP = {};
  TD = {};
  COND = {};
  Foreach sc In SC Do
  Begin
     OP' = OP' U getOper(SC);
     EP = getAllPath(SC);
     TD = createTestData(OP');
     Foreach ep In EP Do
        TD' = TD' U  getTestData(ep, TR);
     TD' = ProcessUnsed(TD);
  End
    Foreach op in OP Do
      TC =TC U Generation (OP);
  TCS = Combine(TC,TD')
End
```

C. Example

Here, we give the actual example to demonstrate the experimental result. The business process consists of four operations, Oa-Od; four conditions, C1-C4. Therefore, the maximum path is 2*2*2*2=16, if consider the possibility of execution, the all possible paths are {OaC1C2Ob, OaC1C2Oc, OaC1C3Oc, OaC1C3C4Oc, OaC1C3C4Od}.

The Oa-Od is needed to design the test item corresponding to an operation.

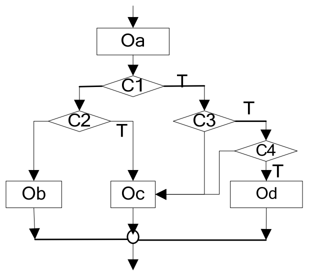


Figure 7. An actual business process

The second step is to design the test data for possible path. If test requirement is condition/decision coverage, some condition coverage should be deleted or redesigned due to the contraction with test data. In table 4, the 8th test data are unreachable for condition (x>1 and y>10000) true. The condition is false due to y=6000 in No.8. And then, we could design the test data for execution path. So the final test data in No.8 should be x=2, y=20000, z=4.

TABLE IV. CONDITIONS AND TEST DATA

No.	Condition (T)	Test Data			
		x	y	z	T/F
1	x>0 or y>5000	0	3000	0	T
2		2	12000	-2	F
3	y<2000 or z<3		1000	2	T
4			4000	3	F
5	x>1 and y>10000	1	5000		T
6		2	15000		F
7	y/(x+z)>10000	1	20000	1	T
8		2	6000	2	F

V. RELATED WORKS

Software testing, as is mentioned by Zhu [3], can be divided into white-box testing and black-box testing, according to whether or not it involves with code. The white-box testing includes testing based on code and testing based on standard. Well, the black-box testing includes hybrid test, which is based on standard. Moreover, FSM (finite state automata) is used to generate test data for lots of models [4-6]. Also, many researches are aim at generating testing path of EFSM (extended finite state automata). As for EFSM testing, testing coverage will involve many aspects, for example, state coverage, transition coverage, path coverage, and so on. These coverages are used to generate FTP (feasible transition path), which is required by test cases [7] [8].

Weyuker's axiomatic system proposed basic properties of software test adequacy criteria. By using axioms set, it could assess the adequacy criteria for the testing. In the evaluation of axiom, Weyuker's criteria for testing are compared with test adequacy of the criteria [2]. The anti-composition axiom in [2] is not positive. This axiom points out, even though test case suite T is adequate for each component in testing program P, T in the case of P, it may not necessarily be adequate. This axiom shows, as for the tested program, the adequacy of testing is impossible to be fulfilled. Although the axiom indicates an intuitive concept in software testing, it cannot enhance the confidence of test engineer with certain test case. So, the test criterion, which is described by the axiom, has a kind of a negative character.

VI. CONCLUSION AND FUTURE WORK

The paper gives the method of generating the complete test case suite according to the business model for the scenario test. The framework is fulfilling the demand of test requirement and supporting the design test case and test suite effectively in scenario testing.

Future work of this research includes deeply research further improve the efficiency and correctness of the test case suite and give more extension of choosing better test cases from the alternative test case which generate thought reuse technique. The related work and the newly exploration of reusing technique are still ongoing for software testing.

ACKNOWLEDGMENT

The work is supported by National Torch Program under Grant No. 2009GH510068 and Shanghai STCSM Program under Grant No.10DZ2291800. The authors would like to thank without knowing the name for their helpful suggestions.

REFERENCES

[1] W. K. Leow, S. C. Khoo, and Y. Sun, "Automated generation of test programs from closed specifications of classes and test cases", Proceedings of the International Conference on Software Engineering, 2004, pp. 96-105.

[2] E.J. Weyuker, Axiomatizing software test data adequacy. IEEE Trans. on Software Engineering, 1986, vol. 12(12), pp. 1128-1138.

[3] H. Zhu and Z. Jin, Software Quality Assurance and Testing. Science Press, Beijing, 1997, pp. 142-147.

[4] Z. Liu, G. Yang, and T Li, "A Component-based Reuse Technique of Software Test Cases, Proceedings of the 3rd World Congress for Software Quality", Munich, Germany, 2005, vol. 1, pp. 26-30.

[5] A. A. Andrews, J. Offutt, and R. T. Alexander, Testing Web Applications by Modeling with FSMs. Software and Systems Modeling, 2005, vol. 4(3), pp. 326-345.

[6] R. Lai, "A survey of communication protocol testing," Journal of Systems and Software, 2002, vol. 62, pp. 21-46.

[7] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," Proceedings of the IEEE, 1996, vol. 84, pp. 1090-1123.

[8] A. Kaliji, R. M. Hierons, and S. Swift, Generating Feasible Transition Paths for Testing from an Extended Finite State Machine(EFSM) In proceedings of 2nd International Conference on Software Testing Verification and Validation, Denver, USA, 2009, pp. 230-239.

A. Y. Duale and M. U. Uyar, "A method enabling feasible conformance test sequence generation for EFSM models," IEEE Transactions on Computers, 2004, vol. 53, pp. 614-627.

Is Mutation Testing Scalable for Real-World Software Projects?

Simona Nica, Franz Wotawa
Institute for Software Technology
Graz University of Technology
Graz, Austria
snica,wotawa@ist.tugraz.at

Rudolf Ramler
Software Competence Center Hagenberg GmbH
Hagenberg, Austria
rudolf.ramler@scch.at

***Abstract*—A significant amount of research has been conducted in the area of mutation testing. It is a fault based technique that has been intensively used, over the last decades, as an efficient method to assess the quality of a given test suite. In the literature different mutation tools are available, corresponding to different programming languages or different types of applications. Although mutation testing is a powerful technique, limitations do exist. The most common problems are represented by the increased computation time, necessary to derive the entire mutation testing process, and the equivalent mutants problem. Therefore a natural question arises: is mutation testing really suitable in real-world environments? Through the research we start here, we aim to come with an accurate answer to this question.**

***Keywords*-mutation testing; mutation tools; coverage tools; eclipse project;**

I. Introduction

Mutation testing is a test technique that has been used to evaluate the test suite of an application, but also for the test case generation process. It is a fault based technique that makes use of a well determined set of faults for measuring the efficiency of the test suites. The mutation process involves the following steps:

1) Faults are introduced into a program resulting in different faulty versions (mutants) of this program.
2) Each mutant is run against the provided set of test suites. When a mutant fails to pass a test case, it is said that the mutant is killed. Otherwise it is still alive or it could not be detected - e.g., due to dead code or because it is an equivalent mutant.
3) The mutation score (the ratio between the number of killed mutants and the number of all mutants) is computed. The mutation score is an indicator used to evaluate the effectiveness of a test suite, i.e., its capability to detect the faults introduced through mutations, and thus describes the test suite adequacy.

A mutant is said to be equivalent with the original program when there is no way that a test case can detect the modification - since the output will always be the same with the output of the original program. Figure 1 presents an example, the arithmetic operator replacement (AOR) mutation. It is important to detect and avoid equivalent mutants because they cause an artificially low mutation score, as they cannot be killed.

Mutation testing is seen as a good metric for measuring the coverage levels achieved through different test coverage techniques. The authors in [1] prove that in some situations coverage measure techniques do not represent the most adequate measure in discovering all the faults an application is prone to. For example, in the case of test driven development, one first writes the tests and then starts writing the source code. In most of these situations the programmer obtains a good coverage of the code, but only those specific faults may be detected, the ones the programmer had thought of during the development of the tests. In contrast, mutation testing can be taken as a good indicator for measuring the coverage levels achieved through different test coverage techniques.

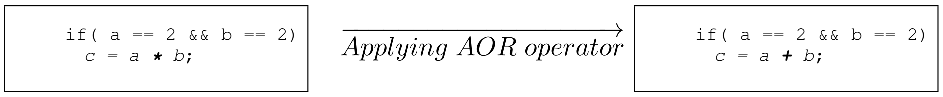
In 1971, Richard Lipton introduced the concept of mutation testing. The technique was further developed by DeMillo, Lipton and Sayward [2]. The technique can be applied at unit testing level [3], [4], [5], integration testing level[6], [7] or it can be used to validate the specifications [8], [9], [10], [11]. Several mutation testing tools were developed, for different existing programming environments: Fortran [12], [13], Java [14], [15], [16], [17], [18], C# [19], [20], C [21] and SQL [22]

Although the mutation testing technique can be computationally very expensive and also time consuming, it has been shown that mutation testing is stronger than coverage based metrics [4]. Therefore a natural question arises: is mutation testing worth the effort in a real life software project?

This paper is structured as follows. In Section 2 we give a brief description of the working environment and the tools used in our research. In Section 3, we present and discuss the results. In Section 4 we discuss the related research. Finally, in Section 5 we conclude the paper.

II. Environment Setting

In this paper, we aim to assess the costs of applying mutation testing on a real-life software system. Following aspects have been investigated to answer the questions whether mutation testing worth the effort in a real life software project:



```
if( a == 2 && b == 2)
  c = a * b;
```

```
if( a == 2 && b == 2)
  c = a + b;
```

Figure 1. Equivalent Mutant

- The time required for mutation testing,
- The results of mutation testing compared to coverage analysis,
- The issues encountered in setting up and running selected mutation testing tools.

In what follows, we briefly present the working environment configuration.

A. Environment Configuration

We have chosen to use mutation testing on Eclipse [23], a widely known and large open source project that shows many parallels to commercial and industrial software projects, especially those developed on the basis of the Eclipse application framework. We retrieved the source code Eclipse Release Build 3.0, from the Eclipse repository [23].

In Table I, the versions and configuration parameters of the tools and test objects used throughout our research are described. All of the presented work was conducted using the virtual environment Oracle VM Virtual Box. The virtual machine is configured to run on Windows XP SP2 operating system, on an Intel Core 1.73 GHz with 2 GB of RAM. For the Java Virtual Machine, we compile and run all the files involved in the research with version 1.6, update 24. We have chosen to work within a virtual environment, in order to offer a fast portability and also an easy management for our research. We aim at a fully automatized process, for all the Eclipse plug-ins, which will run over a predefined period of time, on different architectures.

We apply three of the most widely used mutation tools: MuJava [3], Jumble [16] and Javalanche [15]. We run the mutation testing technique and then compare the results with the code coverage information provided by Clover [27] and EclEmma [28].

B. Applied Mutation Tools

For computing the mutation score metric, we take into account, throughout the research, the following mutation tools:

Tool / System	Version	Location/Comment
Eclipse	3.0	[23]
MuJava	3	[24]
Jumble	1.1.0	[25]
Javalanche	0.3.6	[26]
Clover	3.0.2	[27]
EclEmma	1.5.1	[28]

Table I
OVERVIEW ENVIRONMENT CONFIGURATION

1) **MuJava**: MuJava is a Java based mutation tool, which was originally developed by Offut, Ma, and Kwon [14]. Its main three characteristics are:
   - Generation of mutants for a given program.
   - Analysis of the generated mutants.
   - Running of provided test cases.

   Due to the newly implemented add-ons, the tool supports a command line version for the mutation analysis framework, which offers an easy integration into the testing or debugging process. Offutt proved that the computational cost for generating and executing a large number of mutants can be expensive, and thus he proposed a selective mutation operator set that is used by the MuJava tool. It works with both types of mutation operators:
   - Method level mutation operators (also called traditional), which modify the statements inside the body of a method;
   - Class level mutation operators, which try to simulate faults specific to the object oriented paradigm (for example faults regarding the inheritance or polymorphism).

   MuJava was not designed to work with JUnit test cases, nor to compile with Java versions greater than 1.4; i.e., Java development kit 1.5 or 1.6. Due to the fact that for most of the applications we use throughout the research, we work with MuJava as the mutation testing tool, we have implemented different add-ons to support JUnit test cases and partial mutation of Java source files compiled with JDK 1.5 or greater. We take into account both the traditional mutation operators, i.e., the method level, and the class level ones. MuJava comes with a graphical user interface.
2) **Jumble** It is a class level mutation tool. Moreover, this tool supports JUnit 3 and, recently, it was updated to work with JUnit 4. Similar to MuJava, just one mutation is possible at a time, over the source code under test. First, the tool runs all the tests on the original, unmodified, source file and checks whether they pass or not, recording the time necessary for each test. Then, it mutates the file according to different mutations operators and runs the tests again. The process is done when all the mutations have been tested. Unlike MuJava, Jumble is able to mutate constants.
3) **Javalanche** This mutation testing tool should resolve two major problems in mutation testing: efficiency and equivalent mutants problem. Javalanche works on byte code and can mutate very large programs. The

authors resolve the problem of equivalent mutants by assessing the impact of mutations over the dynamic invariants [29]. According to the authors of the tool, Javalanche has an unique feature. The tool is able to rate the mutations in accordance with their impact on the behavior of program functions, i.e., the greater the impact of an undetected mutation is, the lower the possibility of an equivalent mutant.

We have chosen to conduct the research using the above described tools, taken into account their usage inside the experiments conducted in the mutation testing area.

III. Results

In this section, we present the first results of our research, by taking into consideration the three aspects that we follow in our research work: time, mutation testing results, using the JUnit tests provided on the Eclipse repository, and finally we describe the issues encountered in setting up and running the different mutation testing tools.

A. Research Procedure

In our research we follow the next steps:

1) Check-out the project from the Eclipse repository;
2) Run the plug-in test cases associated to the checked out project;
3) Download and install the coverage and mutation tools;
4) Set all the necessary class paths for each tool;
5) Run the tools over the original project and record the results. This step is the one that consumes most of the time, i.e., approximately 1 month and a half in case of our chosen plug-in project. This is mainly due to the different compilation exceptions encountered; for the compilation and tools running tasks one human resource was allocated.

As the research procedure is the same for each of the Eclipse plug-ins, we conduct the first research steps with the Eclipse Java development tools Core project. The JDT [30] provides the tool plug-ins that implement the Java IDE, which supports the development of any Java application, including Eclipse plug-ins.

The JDT Core project, *org.eclipse.jdt.core*, has associated three test projects:

1) org.eclipse.jdt.core.tests.builder
2) org.eclipse.jdt.core.tests.compiler
3) org.eclipse.jdt.core.tests.model

B. Time

Concerning the time necessary to derive this research, we have to take into account:

- The time necessary to configure the tools; the effort estimated was of approximately one week;
- The mutants generation time; for the selected plug-in, it took us between 6 to 8 hours, i.e., a full working day;
- The time needed to run the test cases against the set of mutants. This is the most significant one, as we have a huge number of mutants.

C. Mutation Results

For each test project from Table II, we computed the total number of initial test cases $\mathbf{No_{TC}}$, the initial time $\mathbf{T_{orig}}$, in minutes, needed to run the tests, and the success rate $\mathbf{S_{rate}}$ which tells us the percentage of tests that initially passed.

In Table III, we show the detailed mutation testing information for one of the three test projects, *org.eclipse.jdt.core.tests.compiler.regression*. We record the number of generated mutations $\mathbf{No_{Mut}}$, the necessary time for generating all the mutations, $\mathbf{T_{Mut}}$, the mutation score **MS** and the total time for running the tests over the mutants, i.e., $\mathbf{T_{TC_{Mut}}}$. MuJava generated 123 class mutants and almost 31 000 method mutants, in approximately 360 minute, i.e., 6 hours. We estimated the total time for running all the generated mutants; we did not run all the method level mutants, due to the increased time complexity. The average mutation score recorded was around 65%. The computed mutation score, for MuJava, is the average of all the mutation scores computed for each run of the plug-in, in accordance with the selected mutants.

As it can be observed from Table III, we were not able to obtain any mutation points for Jumble and Javalanche. By $\mathbf{T_{No_{TC}}}$ we denote the total number of test cases from a specific test project. In Table IV, we record, the success rate for the three plug-in projects, after running all the test cases from each project, using the coverage tools. Concerning the types of code coverage recorded by the tools we have selected for our research, we know that:

- *Clover* measures statement, branch and method coverage;
- *EclEMMA* computes class, method, statement and basic block code coverage.

In the research conducted so far, we have reported the mutation score to the statement coverage level. Further code coverage measures will be taken into account for the mature stages of our research.

D. Encountered Issues

Up to now we were not able to generate mutants, for the JTD Core project, with Jumble or Javalanche; this part of

Test Project	$\mathbf{No_{TC}}$	$\mathbf{T_{orig}}$	$\mathbf{S_{rate}}$
org.eclipse.jdt.core.tests.builder	79	161.312	100.00 %
org.eclipse.jdt.core.tests.compiler	2542	16.203	100.00%
org.eclipse.jdt.core.tests.regression	2622	387.735	100.00%
org.eclipse.jdt.core.tests.eval	350	65.562	100.00%
org.eclipse.jdt.core.tests.dom	1584	136.437	100.00%
org.eclipse.jdt.core.tests.formatter	486	21.109	100.00%
org.eclipse.jdt.core.tests.model	2084	293.782	100.00%

Table II
Eclipse JUnit Test Results

Tool	$\mathbf{No_{Mut}}$	$\mathbf{T_{Mut}}$	MS	$\mathbf{T_{TC_{Mut}}}$
MuJava	123/30947	174.69 min/185.7min	app.65%	est. 2 months
Jumble	-	-	-	-
Javalanche	-	-	-	-

Table III
MUTATION TESTING INFORMATION PER MUTATION TESTING TOOL

Project	$\mathbf{T_{No_{TC}}}$	Clover	EclEMMA
org.eclipse.jdt.core.tests.builder	79	100.00%	100.00%
org.eclipse.jdt.core.tests.compiler	16287	100.00%	100.00%
org.eclipse.jdt.core.tests.model	8639	99.97%	100.00%

Table IV
SUCCES RATE

our work is still in progress. The main problem we have encountered was to run the test cases as plug-ins test, using the different mutation tools. Besides time consuming, the generation of mutants proved to be also very complex.

Concerning the first mutation tool, MuJava, there are some limitations we have to take into consideration:

- MuJava is not able to generate any mutants in case of constants (it does not mutate constant values);
- Also, missing statements are another limitation of the tool. We are not able to generate mutants, by statement deletion nor insertion;
- In case of multiple bugs in one statement, the MuJava tool is not able to mutate more than one variable or operator per statement and mutant, i.e., each mutant contains only one change when compared with the original program (this limitation is however easy to overcome);
- In order to support execution of JUnit tests, the nullary constructor has to be added to each test class file. Also, the private methods *setUp()* and *tearDown()* must have public access;
- The last problem regarding mutation is that sometimes equivalent mutants are generated.

Regarding MuJava, as it can be already observed from Table III, the majority of mutants was represented by the method ones. From this large pool of traditional mutants, the three most commonly encountered were:

1) AOIS, i.e., Arithmetic Operator Insertion, with 13654 mutants,
2) LOI, i.e., Logical Operator Insertion, with 4698 mutants, and
3) ROR, i.e., Relational Operator Replacement, with 3980 mutants.

Javalanche is of real interest in our approach, as it should deal with the equivalent mutant problem. This would allow us to reduce the high number of generated mutants and thus reduce the effort. Therefore, we further try to run the research and, together with the people involved in Javalanche development, come with a solution.

We have to mention that on small and simple projects, i.e., no more than 200 lines of code and which have the test sets in the same project as the mutated classes, we were able to configure and successfully use with success Jumble and Javalanche.

In what follows, we briefly describe the experience recorded for configuring and running the mutation tools we have used in this paper. We denote by $\mathbf{Tool_{Config}}$, i.e., tool configuration, the knowledge accumulated when configuring all the paths; by $\mathbf{Mut_{Gen}}$ we present the mutants generation step and by $\mathbf{Running_{TC}}$ the observations when running the mutants.

1) ***MuJava***
   - $\mathbf{Tool_{Config}}$: The graphical user interface, but also the command line version, are intuitive and easy to use.
   - $\mathbf{Mut_{Gen}}$: The tool must have access to the class files corresponding to each file to be mutated; also, the user can select which mutation operators to apply, both from the set of traditional mutants and also from the class level ones.
   - $\mathbf{Running_{TC}}$: MuJava requires the tests to have the nullary constructor. Also, the methods setUp() and tearDown() must have public access (default is protected). This was time consuming, as we had to update all the test classes with the nullary constructor and the public access for the two methods.
     Both for generating the mutants and then running them it takes a lot of time. A solution may be the integration into an ant script, which is to be run on a monthly basis without any user interaction.
2) ***Jumble***
   - $\mathbf{Tool_{Config}}$: A readme.txt file is available, where the steps to take are quite easy to follow. Nevertheless, after following the instructions and setting the classpath, we were not able to derive a running configuration.
   - $\mathbf{Mut_{Gen}}$: We were not able to get any mutants, due to execution errors.
   - $\mathbf{Running_{TC}}$: Not reachable.
3) ***Javalanche***
   - $\mathbf{Tool_{Config}}$: The javalanche.xml file has to be copied to the current user directory, where it is located the project to be mutated. Then an easy configuration follows, i.e., change the paths to the installation folder and for the working project into the xml file.
   - $\mathbf{Mut_{Gen}}$: Javalanche instruments the byte code and then needs to take control over the test execution. For test execution Javlanche relies on JUnit test suites. If a test suite is not supplied, Javalanche just mutates the code, but it can not

take over the control of the test execution.

- $\mathbf{Running_{TC}}$: We did not manage to reach this step.

Regarding the two coverage tools we have used, we found it easy to setup and integrate them into a daily ant script, but also as an Eclipse plug-in (both Clover and EclEmma can be used as Eclipse plug-ins).

IV. Related research

As mutation testing has proven to be an efficient technique in assessing the quality of the test pool, the attention was focused on whether or not mutation can be used in large scale software applications. In [4], the authors try to answer this problem by running mutation testing over a set of software programs, written in C language, each the size of more than 200 lines of code, and with a large pool of test cases. All the programs had available a pool of faults. The authors were able to show that, when carefully used, the mutation testing technique can provide good results in fault detection.

In [31], the authors proposed a new mutation testing tool, developed in Java and AspectJ for Java programs. They run a research study on real-world open source Java projects, randomly selected, and compare the results with Jumble and MuJava.

What distinguishes our work from the previous ones, is the fact that we take a huge, well known and widely used software project, i.e., Eclipse, and start to record different software metrics. The most important of them is the mutation score metric. For Eclipse we can track the faults database and therefore derive a realistic and practical report of the mutation testing technique, together with other quality software metrics, in order to depict real software bugs. One of the work we report to is the research conducted by Zeller [32].

V. Conclusion

Mutation testing is an efficient method to detect errors inside the software projects. Unfortunately, the available open source mutation testing tools we have used so far in our research work, have proven to take a lot of time in order to derive all the configuration settings. Although mutation testing can assist in revealing many errors, not all of them represent real actual software failures. The problems mostly encountered with this technique are the complexity to derive the process (as higher the number of generated mutants is, as higher the computation time) and also the equivalent mutant problem.

Each of the above described mutation tools requires different configuration settings. The time effort we have invested just in configuring each tool and then deriving the entire mutation testing technique is now of several months. From the results obtained during the research work, we state that mutation can be regarded as a good software quality metric, but special attention should be given to the drawbacks presented above and, also, to the total amount of time. Meanwhile, setting up the configuration and then running the code coverage tools has proven to be easy to conduct. Based on other previous works, we compare the results given by code coverage with the ones obtained from the mutation testing process.

Through this current research work we start a study, trying to answer the title question: *is Mutation Testing Scalable for Real-World Software Projects?*. We aim to further develop this work, trying also to benefit from the advantage offered by Javalanche: equivalent mutant detection.

Acknowledgment

The research herein is partially conducted within the competence network Softnet Austria II (www.soft-net.at, COMET K-Projekt) and funded by the Austrian Federal Ministry of Economy, Family and Youth (bmwfj), the province of Styria, the Steirische Wirtschaftsfrderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

References

[1] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria,” vol. 32, no. 8, August 2006, pp. 608–624.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Program Mutation: A New Approach to Program Testing,” in *Infotech State of the Art Report, Software Testing*, 1979, pp. 107–126.

[3] Y.S.Ma, J. Offutt, and Y. R. Kwon, “MuJava : An Automated Class Mutation System,” vol. 15, 2005, pp. 97–133.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is Mutation an Appropriate Tool for Testing Experiments?” in *Proceedings of the 27th International Conference on Software Engineering (ICSE’05)*, St Louis, Missouri, 15-21 May 2005, pp. 402–411.

[5] A. J. Offutt, “A Practical System for Mutation Testing: Help for the Common Programmer,” in *Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years*, 2-6 October 1994, pp. 824–830.

[6] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, “Interface Mutation: An Approach for Integration Testing,” vol. 27, no. 3, 2001, pp. 228–247.

[7] U. Praphamontripong and A. J. Offutt, “Applying Mutation Testing to Web Applications,” in *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION’10)*, Paris, France, 6 April 2010, pp. 132–141.

[8] W. Krenn and B. Aichernig, “Test Case Generation by Contract Mutation in Spec#,” in *Proceedings of Fifth Workshop on Model Based Testing (MBT’09)*, York, UK, March 2009, pp. 71–86.

[9] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero, "Mutation Testing Applied to Validate Specifications Based on Statecharts," in *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, Florida, 1-4 November 1999, pp. 210 –219.

[10] V. Okun, "Specification Mutation for Test Generation and Analysis," PhD Thesis, University of Maryland Baltimore County, Baltimore, Maryland, 2004.

[11] W. Ding, "Using Mutation to Generate Tests from Specifications," Master Thesis, George Mason University, Fairfax, VA, 2000.

[12] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford, "The Mothra Tool Set," in *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences (HICSS'22)*, 3-6 January 1989, pp. 275–284.

[13] K. N. King and A. J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," vol. 21, no. 7, October 1991, pp. 685–718.

[14] Y. Ma, A. J. Offutt, and Y. Kwon, "MuJava: a Mutation System for Java," in *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)*, Shanghai, China, 20-28 May 2006, pp. 827–830.

[15] D. Schuler and A. Zeller, "Javalanche: Efficient Mutation Testing for Java," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, Amsterdam, Netherlands, 24-28 August 2009, pp. 297–298.

[16] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. J. Inglis, and M. Utting, "Jumble Java Byte Code to Measure the Effectiveness of Unit Tests," in *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, Windsor, UK, 10-14 September 2007, pp. 169–175.

[17] I. Moore, "Jester - a JUnit test tester," in *Proceeding of eXtreme Programming Conference (XP'01)*, 2001.

[18] PIT Mutation Testing, "http://pitest.org/," 2011.

[19] A. Derezinska and A. Szustek, "CREAM- A System for Object-Oriented Mutation of C# Programs," Warsaw University of Technology, Warszawa, Poland, Technical Report, 2007.

[20] Nester, "http://nester.sourceforge.net/," 2011.

[21] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," in *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, Windsor, UK, 29-31 August 2008, pp. 94–98.

[22] J. Tuya, M. J. S. Cabal, and C. de la Riva, "SQLMutation: A Tool to Generate Mutants of SQL Database Queries," in *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, Raleigh, North Carolina, November 2006, p. 1.

[23] Eclipse, ":pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse," 2011.

[24] M. D. Site, "http://cs.gmu.edu/~offutt/mujava/," 2011.

[25] Jumble, "http://jumble.sourceforge.net/," 2011.

[26] Javalanche, "http://www.st.cs.uni-saarland.de/~schuler/javalanche/download.html," 2011.

[27] Clover, "http://www.atlassian.com/software/clover/," 2011.

[28] EclEmma, "http://www.eclemma.org/," 2011.

[29] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," in *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, April 2010, pp. 45–54.

[30] JDT, "http://www.eclipse.org/jdt/," 2011.

[31] L. Madeyski and N. Radyk, "Judy a mutation testing tool for java."

[32] R. Premraj and A. Zeller, "Predicting Defects for Eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07, 2007, p. 9.

Testing As A Service for Component-based Developments

Hien Le
Department of Telematics
Norwegian University of Science and Technology
hiennam@item.ntnu.no

Abstract— **In this paper, we present an approach to model testing as a service for component-based development. The approach is based on the Service-oriented Architecture in which testing services are modeled using UML collaboration structure to support the validation of components. We categorize two types of components: elementary and composite. Elementary components are non-decomposable and reusable computing units. Composite components are developed by composing existing components, which can either be elementary or composite ones. Our main contributions presented in this paper are: (1) to provide an approach for modeling component testing as a service; and (2) to provide a constructive mechanism for composing testing services. In this paper, testing services for railway control system will be used to illustrate our approach.**

Keywords – software components; component testing; testing as a service

I. INTRODUCTION

A component, in general, may be defined as a reusable software or computing unit [1], which is designed to partially or fully perform specific functionalities invoking through component interfaces. The reusable components are normally verified, validated and stored in a repository. Component-based development is a software development approach in which new components are developed by composing existing components retrieved from the component repository [2] to satisfy new requirements. By this approach, on the one hand, new components and software systems can be rapidly developed [3, 4] while reducing development efforts and costs. On the other hand, however, there are many challenges, for examples, how to ensure that these newly developed components do not posse any unusual behaviors [7, 8, 10] while fulfill the requirements.

Component verification and validation are software development activities whose aim is to ensure that newly created components fulfill the requirements without introducing any emerging or unexpected behaviors [7, 10]. In this paper, an approach to model testing as a service to support the component validation, also known as component testing to guarantee that the component fulfills its expected functionalities when performing in the intended environment [12], is presented. The approach is based on the Service-oriented Architecture in which testing services are modeled and composed using UML collaboration structure to support the validation of components.

As shown in Figure 1, the *ComponentUnderTest* represents the service clients, which are newly developed components. These components must be validated. *TestingServiceProvider* plays the role of the service providers, i.e., providing simulation environments and testing suites for validating the new components. *TestingServiceRegistry* is where the descriptions of testing services are published so that they can be found by the service clients, i.e., the *Component Under Test*. When a suitable testing service has been matched with the testing requirements of the new components, the validation process (refereed as testing process in this paper) for these new components can be carried out. The testing process, which is modeled and deployed as a service, emphasizes that testing services are independently developed from the component-based development view; and newly testing services can be developed by composing existing testing services in the same manner as service composition [9]. However, to be able to apply the Service-oriented approach for supporting component testing, we must answer two questions: (1) how to model testing as a service; and (2) how to compose testing services, i.e., constructing new testing services as a composition of existing ones. In this paper, we focus our discussion on these two issues.

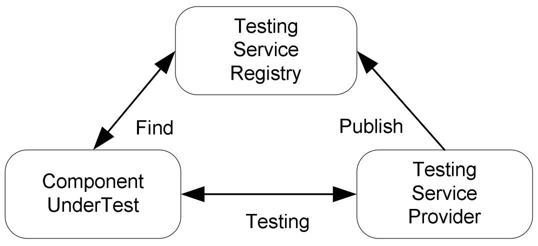


Figure 1: SOA for testing services

In the following discussion, we categorize two types of components: elementary and composite components. Elementary components are the ones which can not be decomposed further. Composite components are composed from existing components, which can be either elementary or composite one. Normally, an elementary component is first designed, verified and validated and stored in a repository to be re-used [4]. The validation of components is ensured by applying test suites to the component interfaces in a simulation environment [11].

The rest of the paper is organized as follows. Related work is discussed in Section II. Section III presents the modeling approach which is based on UML collaboration structure to model testing as a service. Section IV discusses how to create new testing services by service compositions. Conclusion and future works are given in Section V. A railway control system which is built by component-based development approach will be used to illustrate the applicability of our testing service modeling approach.

II. RELATED WORK

In this section, we discuss the related work on modeling testing as a service for components and how to compose testing services. To our knowledge, there are many approaches that support the validation of elementary components [13, 14]. However, the current research which focuses on validating of composite components is very limited [7, 8]. These existing approaches mainly focus on testing specification [14], generating test cases for component testing [7] or performance [11, 13]. Furthermore, these testing approaches do not differentiate the different between elementary and composite components. In [11], a testing method which utilizes the Service-oriented architecture to support testing of complex and safety-critical systems is presented. However, this testing approach focuses on the distribution and performance of testing process, e.g., distributed testing among testing hosts, rather than how to model testing as a service. Existing approaches for designing test suites of elementary components may not be applicable to composite components due to, for example, the new dependencies between sub-components which are the results of composed behaviors of components. Furthermore, the question of how to re-use the test suites or simulation environments, which have been used to validate the elementary components, in the new testing services for composite components may not be fully addressed.

In our recent research [15], a service can be defined as "*an identified functionality aiming to establish some desired effects among collaborating entities*". We have also shown that, based on the collaborative service models, reusable components can be automatically synthesized and such components can then be composed together [16]. Based on this approach, we argue that testing can also be modeled as a service, whose desired goal is to validate the behavior of components, i.e., the two collaborating entities are the component under test and the testing component. From the service models and choreography models of testing services, testing components will be generated and deployed for testing process. Our approach presented in this paper does not focus on issues related to generate test suites for component testing or testing specification (e.g., TTCN-3 [14]), but contributes to modeling testing as a service at abstraction level and to support composition of testing services. This way, the testing of components can be specified at the early phase in the component development lifecycle [2].

III. MODELING TESTING AS A SERVICE

In this section, we first present a railway control system, which is built using a component-based development approach. Second, we will discuss how to model testing as a service for component testing.

A. Train control scenario

Figure 2 shows the overview of the train control system, which is modelled using UML collaboration structure. The operation of the train control system is described as follows. While moving in a geographical region, a *Train* must always be supervised by the *Train Controller Center* (TCC). The TCC responsibility is to monitor and control all train movements in a region.

- The train position on the railway track system is always monitored by the *TCC*. The train, while moving, keeps sending its position report to the TCC. This is modeled as collaboration activity between the *Train* and the *TCC* (i.e., the *PositionReport* collaboration shown in Figure 2).
- The TCC validates the received position information of the train and will issue successive movement authorities (MA) to the train. The MA specifies a safe distance that the train can travel. This is modeled by the *Movement Authority* collaboration.

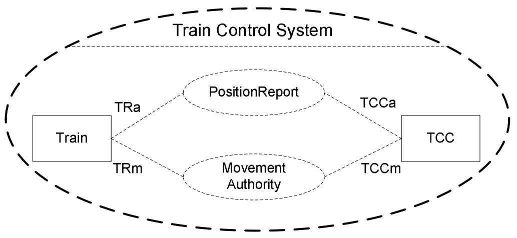


Figure 2: Collaboration structure of the train control system

Based on the collaboration models, the service models and the behavior models of the train control system will be developed and finally the components of the train control system will be synthesized [15, 16]. Figure 3 shows the architecture overview of components of the train control system. The train control system will have the following components.

- The *Position Report* component, which is a sub-component of the *TrainMovementControl* component, reads the location of the train from the external environments, i.e., location indicator installed on the railway tracks [5], and sends this information to the *TCC* component at the control center.
- The *Movement Authority* component handles the movement authority, which is send by the *TCC* to the train. The *Position Report* and the *Movement Authority* components also collaborate to ensure that the train will not travel beyond the safe distance.

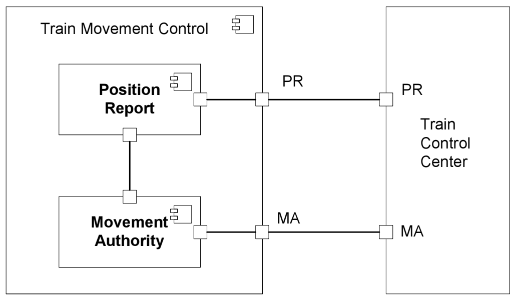


Figure 3: Component view of the train control system

In order to validate the behavior of the *TrainMovementControl* component, which is composed from the *PositionReport* and *MovementAuthority* components, the developer must carry out the following component testing:

- Testing of sub-components: in this case, both the *Position Report* and *Movement Authority* components must be fully tested. The testing of sub-components may in addition require several simulation modules or components [11] which represent the external environments, e.g., location indicators on the railway track systems.
- Testing of the composite component: in this case, the behavior of the composite *TrainMovementControl* component must be verified and validated. In order to validate the *TrainMovementControl* component, the *TCC* counterpart must be available. By our approach to model testing as a service, the corresponding *TCC* will be replaced by a testing component, whose behavior is equivalent to the real *TCC* component (i.e., the *TrainControlCenter* component as shown in Figure 3) during the testing process.

In order to support the testing process, a testing service for components must first be modeled and developed. Next, we present the approach to model testing as a service for components.

B. Testing service for components

The objective of the testing service for components is to support the validation of components at the early stage of development, i.e., design step. Our testing service is based on the concepts of services in which services are defined as a collaboration activity among entities to achieve service goals [6, 15]. Figure 4 shows the basic service structures of the testing service for components.

As shown in Figure 4, the testing service has two main structures, which are specified based on the UML collaboration structure [5], *Simulating* and *Inspecting*. The objective of the *Simulating* is to provide a structural view if the component under test (CUT) requires additional simulation modules. The *Component* role represents the component under test (CUT), and the *EnvSimulator* represents the simulation environment which is required so that thorough test on the component can be performed. The *Inspecting* structure presents the actual testing activity applied on the component, i.e., test suites execution via the *Inspector* role.

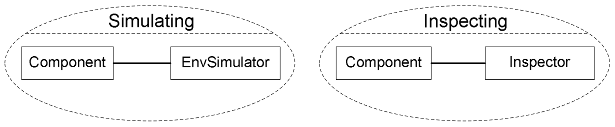


Figure 4: Testing service structures for components

Figure 5 shows the structure of the *Testing Service*, which is the composition of the two testing services, i.e., the *Testing Service* is the composition of *Simulating* and *Inspecting*. The *Testing Service* collaboration includes two main roles: the *ComponentUnderTest* (CUT) role and the *Tester* role. When the testing is performed, the role *ComponentUnderTest* will be dynamically binding to the actual component which will be tested. The main operation of the *Tester* role is to play the role of the testing component which includes the environment simulator (i.e., *EnvSimulator* role) and generated test suites, i.e., to submit test cases to the *ComponentUnderTest* via the *Inspector* role in an intended operation environment. In other words, the *Tester* will implement the interface of the complement testing component. Based on this model, we can identify the structure and specify the test services which take into account the correlation between the required simulation modules and test cases executors.

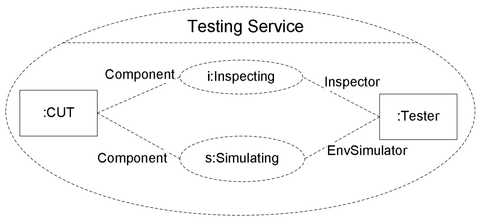


Figure 5: Test model for components

Figure 6(a) illustrates how the *Testing Service* is applied for testing the *Position Report* component. The role *CUT* of the *Testing Service* will be performed by the *Position Report* component, and the *Tester* role will be executed by the *PR_Tester* component, whose functionalities includes both the environment simulation and inspector. Figure 6(b) shows the involved components in the testing process: the *Position Report* is the developed component, and the *PR_Tester* component is synthesized from the testing service model.

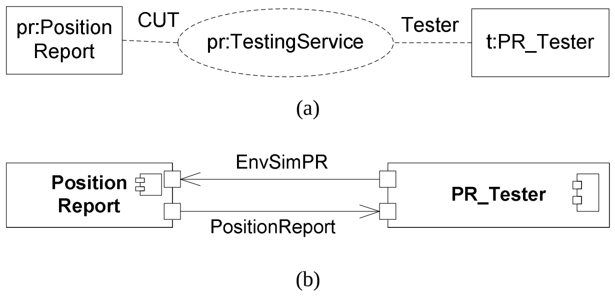


Figure 6: Testing service for Position Report component

IV. COMPOSITION OF TESTING SERVICES

In this section, we present our approach to create testing services which are applied to composite components. In this approach, we discuss an integrated testing service to generate required composite testing services which are composed based on the existing testing services (i.e., of existing components). To simplify our discussion without losing the general discussion details, we assume that all the sub-components of the train control systems have been verified and validated.

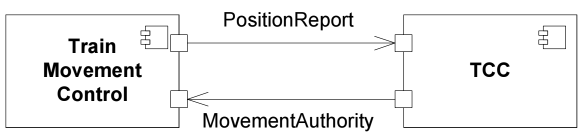
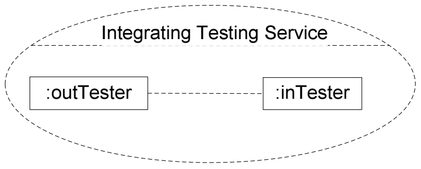


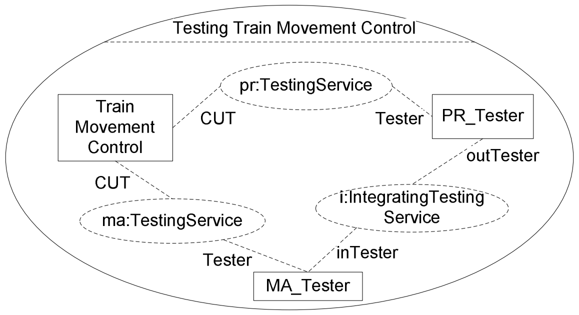
Figure 7: Composite component testing

A. *Integrating testing services for composite components*

As described in Section III, based on the information of the position of the train, the *TCC* will issue movement authority to the train so that the train can safely continue to travel. This means that, for testing the composite component *TrainMovementControl*, the *Tester* role now will be performed by the composite testing component *TCC* which includes both *PR_Tester* and *MA_Tester* roles (as shown in Figure 7). In other words, the output of the *PR_Tester* testing will be validated before the testing of movement authority functionality, i.e., the *MA_Tester*, can be performed. In order to handle the dependency of testing services, we propose an *Integrating Test Service* which provides a mechanism so that the two sub-roles of the *Tester*, i.e., *PR_Tester* and *MA_Tester*, can collaborate. The structural model of the *Integrating Test Service* is shown in Figure 8(a). There are two main roles: *outTester* and *inTester* which perform the sending results from the previous testing service, i.e., testing of the *Position Report* component, and initiating the next testing service, i.e., testing of the *Movement Authority* component.



(a)



(b)

Figure 8: Integrating testing service for Train Movement Control component

Figure 8(b) shows how the *Integrating Testing Service* is re-used and composed to the composite testing service, explained as follows:

- The *pr:TestingService* collaboration is the original testing service for the *Position Report* component and involves two roles *CUT* and *Tester*.
- The *ma:TestingService* collaboration is the original testing service of the *Movement Authority* component and involves two roles *CUT* and *Tester*.
- The *IntegratingTestingService* is re-used to integrate the two existing testing services *pr:TestingService* and *ma:TestingService*. In this situation, the role *outTester* and *inTester* is binding to the *PR_Tester* and *MA_Tester*, respectively.

There are several advantages of our *Integrating Testing Service*. First, the testing service provides a flexible mechanism to support the integration of testing services which have been applied to existing components. Second, the integrating test service focuses on describing the integration of testing services at the design stages while components are being developed. This ways, the testing of composite component can be early specified and carried out.

B. *Realization and deployment of testing services*

The *Integrating Testing Service* provides a mechanism for composing testing services for composite components. This testing service can be deployed in either centralized or distributed testing systems. For example, a centralized testing system can be deployed if both *outTester* and *inTester* roles are realized, i.e., implemented, as testing sub-components of the *Tester* component. In other words, the *Tester* will now perform both *PR_Tester* and *MA_Tester* roles. Figure 9 illustrates a distributed testing scenario in which the sub-components *Position Report* and *Movement Authority* are tested in different systems. In this case, both the distributed testing sub-components *PR_Tester* and *MA_Tester* must implement the *Integrating Testing Service* interface, i.e., *outTester* and *inTester* roles, respectively.

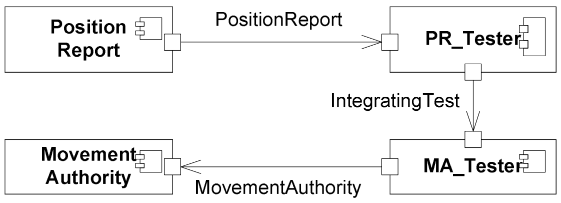


Figure 9: Distributed testing scenario

V. CONCLUSION AND FUTURE WORK

In this paper, we have presented an approach to model testing as a service for component-based development approach. An *Integrating Testing Service* which supports the composition of testing services, i.e., to support the integration and re-usability testing services of existing components, is also presented. This ways, new testing services for composite components can be quickly composed and deployed in either centralized or distributed testing systems.

In future work, we plan to further using the Model-Driven Development approach to automatically synthesize the testing components. A full testing framework, which includes both service models [15] and component-based approach [16], can be developed to dynamically discover and compose for testing of composite components.

REFERENCES

[1] Clemens Szyperski. Component Software: Beyond Object- Oriented Programming. Addison-Wesley Longman Publish- ing Co., Inc., Boston, MA, USA, 2002.

[2] Ivica Crnkovic, Stig Larsson, and Michel R. V. Chaudron. Component-based Development Process and Component Lifecycle. CIT 13(4), 321-327, 2005.

[3] Jisa Dan Laurentiu. Component based development methods: comparison, Computer systems and technologies, 1-6, 2004.

[4] Kung-Kiu Lau and Zheng Wang. Software Component Models. IEEE Trans. Software Eng. 33(10): 709-724, 2007.

[5] Surya Bahadur Kathayat, Rolv Bræk, and Hien Le. Automatic derivation of components from choreographies - a case study. International conference on Software Engineering, 2010.

[6] Surya Bahadur Kathayat and Rolv Bræk. From flow- global choreography to component types. In System Analysis and Modeling (SAM), LNCS 6598, 2010.

[7] Camila Ribeiro Rocha and Eliane Martins. A Method for Model Based Test Harness Generation for Component Testing. 14(1): Journal of the Brazilian Computer Society (JBCS), 7-23, 2008.

[8] Gerardo Padilla, Carlos Montes de Oca, and Cuauhtemoc Lemus Olalde. An Execution-Level Component Composition Model Based on Component Testing Information. 10th International Symposium on Component-Based Software Engineering, 2007.

[9] Surya Bahadur Kathayat, Hien Le, and Rolv Bræk. A Model-Driven Framework for Component-based Development, SDL forum 2011 (to appear).

[10] Jerry Gao and Ming-Shih Shih. A Component Testability Model for Verification and Measurement. International Computer Software and Applications Conference (COMPSAC), 2005.

[11] Renato Donini, Stefano Marrone, Nicola Mazzocca, Antonio Orazzo, Domenico Papa, and Salvatore Venticinque. Testing Complex Safety-Critical Systems in SOA Context. International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), 2008.

[12] 7CMU/SEI-2000-TR-028, CMMISM for Systems Engineering/Software Engineering, Version 1.02, Software Engineering Institute, 2000.

[13] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. Operating Systems Review 43(4), 5-10, 2009.

[14] Bernard Stepien, Liam Peyton, and Pulei Xiong. Framework testing of web applications using TTCN-3. Journal on Software Tools for Technology Transfer (STTT), 10(4), 371-381, 2008.

[15] Surya Bahadur Kathayat, Hien Le and Rolv Bræk. Collaboration-based Model-Driven Approach for Business Service Composition. Book chapter in Handbook of Research on E-Business Standards and Protocols: Documents, Data and Advanced Web Technologies, Ejub Kajan, Frank-Dieter Dorloff, Ivan Bedini, IGI, 2011 (to appear).

[16] Surya Bahadur Kathayat, Hien Le and Rolv Bræk. A Model-Driven Framework for Component-based Development, SDL 2011 (to appear).

A Zone-based Reachability Analysis of Variable Driven Timed Automata

Omer Nguena-Timo
University of Bordeaux (LaBRI - CNRS)
Talence, France
nguena@labri.fr

Antoine Rollet
University of Bordeaux (LaBRI - CNRS)
Talence, France
rollet@labri.fr

***Abstract*—In this paper, we propose an algorithm for efficient reachability analysis of Variable Driven Timed Automata (VDTA). VDTA is a new timed behavioural model for data-flow reactive systems in which inputs and outputs are modelled by variables. Such an approach is commonly used in the industry. Reachability analysis is often the basis of model-checking, test generation (especially Test Purpose approach), or control algorithms. For example, a model-based testing framework with VDTA derives test cases by performing a region-based reachability analysis. Thus, an efficient analysis is needed. We propose an algorithm based on the zone abstraction. The algorithm not only checks the reachability, but it also computes timed sequences of input updates required to reach a target. In practice, it is more efficient than region based one.**

***Keywords*-reachability analysis, zone, timed systems, data-flow, urgent edges.**

I. Introduction

In the context of reliable systems design, formal methods provide a rigorous framework for modelling and reasoning about the systems. Formal reasoning includes methods for testing [15], model-checking [9], supervising [14], etc., the systems, which are commonly and efficiently modelled in a state-transition/automata formalism in which systems behaviours are represented by sequences of (constrained/guarded) transitions between states. Reachability analysis amounts to checking whether a target state is reachable from an initial one. A target state may represent a failure of the system that may cause considerable damages. Reachability analysis is often involved in many validation methods. For example, the generation of test cases with test purposes usually consists in analysing the reachability of certain states and to compute the sequence of actions permitting to reach them. The paper presents a new efficient algorithm for the reachability analysis of Variable Driven Timed Automata [12].

A. Variable Driven Timed Automata (VDTA)

VDTA [12] is a new timed model adapted for modelling and reasoning about data-flow reactive systems in which input and output are rather modelled by variables. A VDTA model-based testing framework has been developed in [11], [13].

A VDTA is a guarded edge (transition) system in which every edge is labelled with an update of output variables and a constraint on input, output and clock variables. VDTA is inspired by urgent timed input/ouput automata [2], [3]. VDTA implements three main mechanisms. The first mechanism, which is not new, implemented in VDTA is *urgent edges*. As with urgent timed automata [3], edges in VDTA are urgent meaning that they are fired as soon as their constraints become true. Only output variable updates are performed on edges firings. It is well-known [3] that urgency mechanism may allow short and clear specification. Secondly, VDTA implements the *variable based communication mechanism*. This supposes that systems communicate with their environment through input and output variables or sockets. This is closer to how engineers think and how open systems are specified. The VDTA model assumes that the environment freely updates the input variables and only the systems can update the output variables according to their states and timing information given by clock variables. VDTA allows to specify explicitly the events to the environment only. Unlike [2], [3], [8], the events from the environment are not explicitly specified. Thirdly, VDTA is a *variable driven model*. Edges firings in VDTA do not depend on occurences of synchronizing actions but rather on the truth value of constraints only. The values of the variables are persistent and they last until they are updated. Consequently, a single input variable change can trigger instantaneously a chain of consecutive edges in VDTA. This is not the case with event-based formalisms like [3], [8] where each edge firing is provoked by an occurrence of a (non persistent) synchronising action.

The VDTA-based testing method [11], [13] is based on test purposes. Test purposes are VDTA with special states labelled with ACCEPT; they allow to guide the test selection. Roughly speaking, the method consists of two main steps: (1) a test graph is constructed from the specification and a test purpose and then (2) the reachability of the states labelled with ACCEPT is checked. A VDTA reachability analysis algorithm is proposed in [11]. This algorithm is inspired by a seminal reachability algorithm for timed automata [2]. It uses the symbolic region [2] representation of the time, which allows elegant reasoning on timed systems. Since it is a precise discrete representation of clock valuations, it allows to abstract a timed model into a discrete model making thus easier the analysis of urgent edge and unspecified input updates in VDTA. In the backside, region abstraction is expensive. As for the analysis of timed automata [4], [7], we wondered how the symbolic zone abstraction could be used for the reachability analysis of VDTA. Roughly speaking, zones are larger abstractions of clock valuations: one zone can be decomposed into an equivalent finite set of regions. Thus a single computation over a zone may need many computations over the regions it includes. Consequently, the region abstraction is in practice less efficient than the zone abstraction [2], [6]. So, providing an efficient zone-based analysis for VDTA will improve the VDTA test generation algorithm [11], [13].

B. Contributions

We propose a backward zone-based reachability analysis algorithm for VDTA. The general idea is presented in Algorithm 1. Contrary to the forward one, it starts in the target states P_0 and iteratively visits *predecessors* of states until a fixpoint is reached and no new states is computed. P_0 is reachable if the fixpoint include the initial state.

Algorithm 1 Principle of the Backward Analysis

```
$P_0 \leftarrow P$
**repeat**
    $P_{i+1} \leftarrow P_i \cup Predecessor(P_i)$
**until** $P_{i+1}$ equal to $P_i$
```

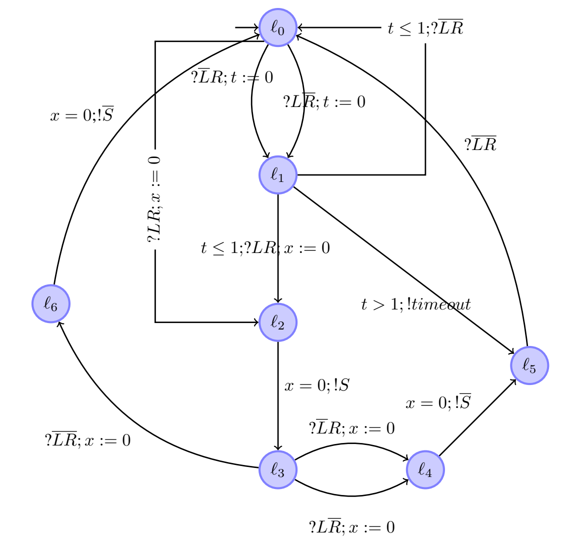
The predecessors of a state of a VDTA include its *input updates predecessors*, *output updates predecessors* and *time predecessors*. To ensure its termination, our algorithm rather works on symbolic states. Input, output and clock values are represented by zones in symbolic states. A similar method is used for timed automata [2], [3], [5], [7]. But the zone-based backward reachability analysis for VDTA could not be a simple adaptation of the zone-based reachability of timed automata. Besides, we found that the analysis of VDTA has many common points with the *analysis of timed game automata (TGA)* [6]. But TGA analysis algorithm cannot work for VDTA. Nevertheless, as for the analysis of TGA we consider a notion of safe time predecessor in order to compute predecessors of symbolic states.

C. Outline

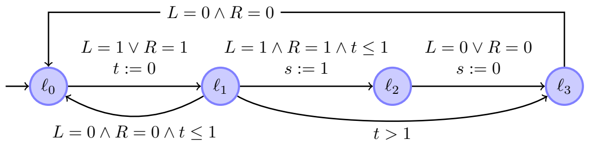
Section II presents the VDTA model and its semantics. In Section III we define symbolic states and the computation of their predecessors. At the end of the section we present the reachability algorithm and we discuss about its termination. Section IV concludes the paper giving future works with VDTA.

II. Variable Driven Timed Automata (VDTA)

Before we provide a formal definition for VDTA, we consider the following specification of the control program designed to start a "two buttons machine" [10]:
The machine starts when two buttons (L and R for left and right buttons) are pushed within 1 time unit. If only one button is pushed (then L or R is true) and a delay of greater than 1 time unit is performed (time-out has occurred), then the whole process must be started again. After the machine has started (S=1), it stops as soon as one button is released, and it can start again only after both buttons have been released (L and R are both false).



(a) TIOA model for the two buttons machine.



(b) VDTA for the two buttons machine.

Fig. 1. Event-based Vs variable-Based Model

The VDTA in Figure 1(b) is clearer and shorter than the Timed Input Output Automata (TIOA) in Figure 1(a). The TIOA has 4 input events (LR, $\overline{L}R$, $L\overline{R}$, $\overline{LR}$ where $L/\ \overline{L}$ mean that the left button is pushed/released), 2 output events (S, $\overline{S}$) and 2 clocks (x, t) whereas the VDTA model has 2 boolean input variables(L and R), 1 boolean output variable (S), and a unique real-valued clock variable (t). The TIOA assumed that in every state the program may receive (denoted by the symbol "?") an event that corresponds to a combination of values of L and R before leaving the state. More generally, if there are n buttons (variables) all in domains of size m, one may consider up to n^m outgoing edges for each location. This explosion can be reduced using a variable-based modelling approach. The main idea is to hide the synchronisation (between the environment and the system) that happens on variable updates and to concentrate on the functional behaviour of the system that depends on constraints. Even if there is not an explicit edge from ℓ_0 to ℓ_2 in Figure 1(b), one can move instantaneously (through ℓ_1) from ℓ_0 to ℓ_2 if L and R are pressed simultaneously. Such a behaviour has required an explicit edge from ℓ_0 to ℓ_2 in Figure 1(a).

However, the VDTA formalism would not have been advantageous in situations where the behaviours depend on former states of the buttons. This may occur when a rising edge of the button needs to be captured and processed even if the button has been released again in the meantime; in such a situation, we would probably need to keep the state "button was pressed" and use a TIOA model.

Anyway, modelling input/output by variables is quite ordinary for engineers. This allows short and clear specification in some circumstances.

A. Model and Semantics of VDTA

Let $\mathbb{N}$, $\mathbb{Q}_+$ and $\mathbb{R}_+$ denote the sets of natural, non-negative rationals and real numbers, respectively. Let $V = \{V_1, \cdots, V_n\}$ be a set of variables; each variable $V_i \in V$ ranges over a (possibly infinite) domain $Dom(V_i)$ in $\mathbb{N}$, $\mathbb{Q}_+$ or $\mathbb{R}_+$. We define $Dom(V) = \Pi_{i\in[1..n]} Dom(V_i)$, the domain of V. In the sequel, v_i denotes a valuation of the variable V_i and v the tuple of valuations of the set of variables V. A variable assignment for V is a tuple $\Pi_{i\in[1..n]}(\{V_i\}\times(Dom(V_i)\cup\{\perp\}))$ and we denote by $A(V)$ the set of variable assignments for V. Given a valuation $v = (v_1, \cdots, v_n)$ of V and a variable assignment $A \in A(V)$, we define the tuple of valuations $v[A]$ as $v[A](V_i) = c$ if (V_i, c) is an element of A and $c \neq \perp$, and $v[A](V_i) = v_i$ otherwise. Intuitively, an element (V_i, c) of variable assignment A, requires to assign c to the variable V_i if c is a constant from $Dom(V_i)$; otherwise c is equal to $\perp$ and *no access* to the variable V_i should be done. $Var(A)$ denotes the set of variables of V that are updated by A. We denote Id_V the identity variable assignment that let unchanged all the variables of V. We denote by $\mathcal{G}(V)$ (resp.$\mathcal{G}^+(V)$) the set of variable constraints defined as conjunction (resp. boolean combinations) of simple constraints of the form $V_i \bowtie c$ with $V_i \in V$, $c \in Dom(V_i)$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $G \in \mathcal{G}^+(V)$ and a valuation $v \in Dom(V)$, we write $v \models G$ when $G(v) \equiv true$ and we define $[\![G]\!] = \{v \in Dom(V) \mid v \models G\}$. For a subset I of V, we denote G_I *the projection* of G over the variables in I. To ease the notation, if I, O, are two sets of different sorts of variables, a constraint G in $\mathcal{G}^+(I, O)$ is a boolean combination of a constraint $G_I \in \mathcal{G}^+(I)$, and a constraint $G_O \in \mathcal{G}^+(O)$. This generalises to an arbitrary number of sets of variables.

Definition 1: A *Variable Driven Timed Automaton* (VDTA) is a tuple $\mathcal{A} = \langle L, X, I, O, \ell^0, G^0, \Delta_{\mathcal{A}}\rangle$, where L is a finite set of *locations*, X is a finite set of clocks, I and O are disjoint finite sets of input and output variables, $\ell^0 \in L$ is the *initial location*, $G^0 \in \mathcal{G}(I, O)$ is the *initial condition* with only one solution, a constraint with variables in $I \cup O$ and $\Delta_{\mathcal{A}} \subseteq L \times \mathcal{G}(I, O, X) \times A(O) \times 2^X \times L$ is the set of *edges*.
In an edge $\langle \ell, G, A, \mathcal{X}, \ell'\rangle \in \Delta_{\mathcal{A}}$ (often written $\ell \xrightarrow{G,A,\mathcal{X}} \ell'$): $G \in \mathcal{G}(I, O, X)$; $A \in A(O)$ is an assignment on output variables and $\mathcal{X} \in 2^X$ is a set of clocks that are reset when passing the edge. There is no explicit assignment on input variables.

A *state* of a VDTA $\mathcal{A}$ is of the form (ℓ, i, o, x) where $\ell \in L$ is a location, i, o and x are *valuations* of input, output and clock variables. A valuation is simply a function that returns the values of the variables. For example, $(\ell_0, (0, 0), 1, 0.5)$ is a state of the VDTA in Figure 1(b) where $(0, 0)$ is the valuation of the inputs L and R, S equals 1 and t equals 0.5.

If $A \in A(I)$ is an assignment on input variables, the valuation $i[A]$ changes the value of input variables according to the assignment. If x is clock valuation, $\mathcal{X}$ is a subset of clocks, and $\delta \in \mathbb{R}_+$ a delay, the valuation $x + \delta$ adds δ to each clock value and the valuation $x[\mathcal{X} \leftarrow 0]$ resets from x all clocks in $\mathcal{X}$. For example, if $i = (0, 0)$ and $A = \{L := 1; R := 1\}$ is an assignment over L and R then $i[A]$ is the valuation $(1, 1)$.

Definition 2: The *semantics* of a VDTA $\mathcal{A}$ is a timed transition system $[\![\mathcal{A}]\!] = \langle S_{\mathcal{A}}, s^0, \Sigma, \rightarrow\rangle$ where $S_{\mathcal{A}} = L \times Dom(I) \times Dom(O) \times \mathbb{R}_+^X$ is the (infinite) set of *states*, $s^0 = (\ell^0, i^0, o^0, x^0)$ is the initial state where x^0 is the clock valuation that maps every clock to 0 and (i^0, o^0) is the only solution of G^0, $\Sigma = A(I) \cup A(O) \cup \mathbb{R}_+$ is the (infinite) set of actions, and $\rightarrow$ is the transition relation with the following three types of transitions:

T1 $(\ell, i, o, x) \xrightarrow{A} (\ell', i, o[A], x[\mathcal{X} \leftarrow 0])$ if there exists $(\ell, G, A, \mathcal{X}, \ell') \in \Delta_{\mathcal{A}}$ such that $(i, o, x) \models G$,

T2 $(\ell, i, o, x) \xrightarrow{A} (\ell, i[A], o, x)$ with $A \in A(I)$ if $\forall(\ell, G, A', \mathcal{X}, \ell') \in \Delta_{\mathcal{A}}$, $(i, o, x) \not\models G$.

T3 $(\ell, i, o, x) \xrightarrow{\delta} (\ell, i, o, x + \delta)$ with $\delta > 0$ if for every $\delta' < \delta$, for every symbolic transition $(\ell, G, \mathcal{X}', \ell') \in \Delta_{\mathcal{A}}$, we have $(i, o, x + \delta') \not\models G$.

The semantics considers discrete transitions (T1 and T2) and continuous transitions (T3). *Output-update transitions* of type T1 allow to update the output variables. Output-update transitions correspond to edges passing. Edges are passed as soon as their constraints are satisfied. *Input-update transitions* of type T2 allow to update the input variables. *Time-elapsing transitions* of type T3 represent the continuous elapse of time.

Definition 3: A *stable state* is a state from which no output-update transition can be fired.

Note that input-updates and time-elapsing transitions are allowed stable states only; they are not allowed in non stable states. Input-update transitions allow to change the inputs. They are fired by the environment. Considering Figure 1(b), $(\ell_0, (1, 0), 0, 0)$ is not stable whereas $(\ell_0, (0, 0), 0, 0)$ is stable. Consequently, the only transition from $(\ell_0, (1, 0), 0, 0)$, $(\ell_0, (1, 0), 0, 0) \xrightarrow{Id_O} (\ell_1, (1, 0), 0, 0)$ corresponds to the edge $\ell_0 \xrightarrow{L=1\vee R=1; t:=0} \ell_1$ $(\ell_0, (0, 0), 0, 0)$ whereas from $(\ell_0, (0, 0), 0, 0)$ one can perform input-update transitions or time-elapsing transitions like: $(\ell_0, (0, 0), 0, 0) \xrightarrow{L=1} (\ell_0, (1, 0), 0, 0)$, $(\ell_0, (0, 0), 0, 0) \xrightarrow{0.3} (\ell_0, (0, 0), 0, 0.3)$ or $(\ell_0, (0, 0), 0, 0) \xrightarrow{0.7} (\ell_0, (0, 0), 0, 0.7)$.

Definition 4: A run of $\mathcal{A}$, $r = s_0 a_1 s_1 \cdots a_n s_n$ in $S_{\mathcal{A}}.(\Sigma.S_{\mathcal{A}})^*$ is a sequence of alternating states $s_i \in S_{\mathcal{A}}$ and actions $a_i \in \Sigma$ with $\Sigma = A(I) \cup A(O) \cup \mathbb{R}_+$ such that $\forall i \geq 0, s_i \xrightarrow{a_{i+1}} s_{i+1}$.

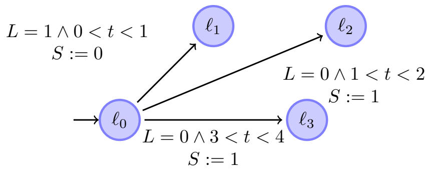


Fig. 2. A VDTA

Here are two possible runs of the VDTA in Figure 1(b): $(\ell_0,(0,0),0,0) \xrightarrow{L:=1} (\ell_0,(1,0),0,0) \xrightarrow{Id_O} (\ell_1,(1,0),0,0) \xrightarrow{0.3} (l_1,(1,0),0,0.3)$ and $(\ell_0,(0,0),0,0) \xrightarrow{L:=1,R:=1} (\ell_0,(1,1),0,0) \xrightarrow{Id_O} (\ell_1,(1,1),0,0) \xrightarrow{s:=1} (\ell_2,(1,1),1,0)$.

B. Principle of Reachability Analysis

The *reachability analysis* amounts to checking whether there exist a run that goes through a given target state. We want that the VDTA reachability analysis algorithm returns how and when to modify the inputs in order to reach the target state eventually.

Let us consider the small running example VDTA in Figure 2 where L is the only boolean input variable, S is the only boolean output variable and t is the only clock variable. Now assume that one wants to check the reachability of the location ℓ_3 whatever are the values of L, S and t in ℓ_3. We want the reachability algorithm to return the verdict "yes" and the following scenario:

1) From the initial state $(\ell_0,0,0,0)$, keep the value of L unchanged and let the time elapse until t equals 1.
2) When t equal 1, set L to 1; then let the time elapse.
3) set L to 0 after the value of t has passed 3 and before it reaches 4.

But if one wants to check the reachability of ℓ_3 when S equals 0, the algorithm should return that the state is not reachable since S equals 1 after the edge from ℓ_0 to ℓ_3 is taken.

Let *Goal* be a set of *states* of $\mathcal{A}$ we want to check the reachability of. Algorithm 1 gives the principle of the backward reachability analysis for $\mathcal{A}$: it starts in *Goal* and computes in each step the predecessors of already encountered states. The algorithm stops when no new state is computed.

a) Predecessors: Let $P \subseteq S_{\mathcal{A}}$ be a subset of states of $\mathcal{A}$. The computation of the set of predecessors of P (denoted $Pre(P)$) involves the computation of its *output-update predecessors* ($Pre_o(P)$), its *input-update predecessors* ($Pre_i(P)$) and its *time predecessors* ($Pre_t(P)$).

The *output-update predecessors* of P, $Pre_o(P)$ is the set of states from which a state in P can be reached just after an output-update is performed; formally, $Pre_o(P) = \{(\ell,i,o,x) \mid \exists\, A \in A(O)\ \exists\, (\ell',i',o',x') \in P \text{ s.t } (\ell,i,o,x) \xrightarrow{A} (\ell',i',o',x')\}$. None of the states in $Pre_o(P)$ is stable.

Similarly, we define the *input-update predecessors* of P, $Pre_i(P) = \{(\ell,i,o,x) \mid \exists A \in A(I),\ \exists\ (\ell',i',o',x') \in P \text{ s.t } (\ell,i,o,x) \xrightarrow{A} (\ell',i',o',x')\}$. All the states in $Pre_i(P)$ are stable.

The *time predecessors* of P, $Pre_t(P)$ is the set of states from which a state in P can be reached by letting the time elapse. Formally, $Pre_t(P) = \{(\ell,i,o,x) \mid \exists\delta > 0,\ \exists\ (\ell',i',o',x') \in P \text{ s.t } (\ell,i,o,x) \xrightarrow{\delta} (\ell',i',o',x')\}$. All the states in $Pre_t(P)$ are stable.

Finally, $Pre(P) = Pre_o(P) \cup Pre_i(P) \cup Pre_t(P)$.

For example, let us consider the VDTA in Figure 2. The state $(\ell_0,0,0,2.2)$ is a time predecessor of $(\ell_0,0,0,2.8)$; but $(\ell_0,0,0,1.999)$ is not a time predecessor of $(\ell_0,0,0,2.8)$ since $(\ell_0,0,0,1.999)$ is not stable and the transition to ℓ_2 is taken prior to time elapsing. $(\ell_0,0,0,1.999)$ is an output-update predecessor of $(\ell_2,0,1,1.999)$.

An execution of Algorithm 1, which computes predecessors of states, may not terminate. This is because VDTA has infinitely many states and each computation step may return a new state. Thus, we consider symbolic representations for states or shortly *symbolic states*; then we describe the computation of predecessor (Pre_o, Pre_i, Pre_t and Pre) of symbolic states. Later, we show that the number of symbolic states encountered during the execution of the algorithm is finite. This will ensure its termination.

b) Relation with other models: A similar zone based analysis method is used for timed automata [2], [3], [5], [7]. But it cannot be simply adapted for VDTA. The output predecessor operator over symbolic states of VDTA works like the discrete predecessor operator for timed automata whereas the input and time predecessors do not. In particular, the computation of the time predecessor should not return time instants that enable an urgent edge. Similarly, the input predecessors can not return input data that enable urgent edges. Besides, the analysis of VDTA has many common points with the *analysis of timed game automata (TGA)* [6]. Indeed, since input-update are freely performed by the environment, they are analogous to "controllable actions" in TGA. Output update transitions are analogous to "uncontrollable actions" since all edges in VDTA (that allow output-update) are eager and fired as soon as their constraints become true. But TGA analysis algorithm cannot work for VDTA: edges passing in VDTA cannot be restricted by letting the time progress forever or by changing the inputs. This is because all edges in VDTA are urgent. Moreover, input actions are not explicit in VDTA whereas they are in TGA. Nevertheless, for the analysis of VDTA we have considered the notion of safe time predecessor inspired from [6].

III. Predecessors of Symbolic States

There are two popular symbolic representations of a set of clock valuations: the region-based [2] and the zone-based [4] representations. Roughly speacking, regions and zones are abstract representations of (infinite) set of valuations; but zones are larger representations: a zone can be decomposed into an equivalent finite set of regions. A region-based analysis algorithm for VDTA inspired by [2] early appeared in [11]. The algorithm is used for selecting real-time test cases for systems modeled with VDTA. The region abstraction allows elegant reasoning on timed systems however it is practically less efficient than the zone abstraction: a single operation over a zone needs many operations over the regions it includes. We consider a zone-based approach for representing symbolic states and to compute their (input/output/time) predecessors.

A. Symbolic States, Zones, and Operations

A symbolic state for $\mathcal{A}$ is a quadruple (ℓ, Z_I, Z_O, Z_X) where $\ell \in L$, $Z_X \subseteq \mathbb{R}_+^X$ is the standard *clock-zone* as defined in [4], $Z_I \subseteq Val(I)$ is a subset of $Val(I)$ such that $Z_I = [\![G]\!]$ for some $G \in \mathcal{G}(I)$; the same for Z_O. We say that Z_I and Z_O are zones over input and output variables. A symbolic state represents a set of states. We write $(\ell, i, o, x) \in (\ell', Z_I, Z_O, Z_X)$ if and only if $\ell = \ell'$, $i \in Z_I, o \in Z_O$ and $x \in Z_X$. A symbolic state is *stable* iff every state in it is stable.

The input, output and clock-zone Z_I, Z_O and Z_X are represented with difference bound matrices (DBM) [4]. DBM are kinds of constraints (comparison between two clocks are allowed). Considering Figure 2, $(\ell_0, L = 1, true, 1 \leq t < 4)$ is one symbolic state where $true$ denotes the constraint that is always true.

Remark: zones for input/output values are simpler than the ones used for linear integer systems or other models with variables (like in [1]). This is because constraints in VDTA never compare clock with input/output variables. Abstractions for clock valuations can be separated for abstractions of signal values. It is not the goal of this paper to discuss the effect of this restriction on the expressiveness of the model.

Remark: in the sequel, constraints are often considered as zones for simplifying the notations. We implicitly assume that every constraint in $\mathcal{G}^+(I, O, X)$ is equivalent to a set of constraints of $\mathcal{G}(I, O, X)$.

Computable operations on zones [4]: if Z is a zone (over clocks or variables), and A is a variable assignment or a clock reset then $[A]Z = \{x \mid x[A] \in Z\}$ denotes the set of valuations from which we can reach a valuation in Z after A is executed. Similarly we define $Z[A] = \{x[A] \mid x \in Z\}$. The *past* of a clock zone Z, $Z{\downarrow} = \{x \mid \exists\ \delta \in \mathbb{R}_+, \text{ s.t } x + \delta \in Z\}$ is the set of valuations (the zone) from which valuations in Z can be reached by letting the time elapse.

Safe time predecessors: given two clock zones Z and g, $Pre_t^s(Z, g)$ denotes the set of *safe time predecessors* [6] of Z with respect to g. Intuitively, a clock valuation x' belongs to $Pre_t^s(Z, g)$ if from x' we can reach $x \in Z$ by time elapsing and along the (time) path from x' to x we avoid all valuations in g. Later, the safe time predecessor is involved in the computation of the time predecessors. Intuitively, it will prevent the time predecessor to return zones that enable urgent edges. Formally:

$$\begin{aligned}Pre_t^s(Z,g) = \{x' \in \mathbb{R}_+^X \mid \exists\ \delta \in \mathbb{R}_+\ x \in Z \text{ s.t } x = x' + \delta,\\ \text{and } \forall \delta' \in [0, \delta], \text{ we have } x' + \delta' \notin g\}\end{aligned}$$

The computation of $Pre_t^s(Z, g)$ is effective [4], [6]: $Pre_t^s(Z, g) = (Z{\downarrow} \cap (\neg(g{\downarrow}))) \cup ((Z \cap (g{\downarrow}) \cap \neg g){\downarrow})$ where Z, g are convex sets and $\neg g$ is the complement of g.

Partitioning of set of zones: later, we will also need to split sets of zones into equivalent sets of disjoint zones. Indeed, since zones are abstractions of larger sets of valuations, it could happen that the input/time predecessors of two valuations in a same zone differ according to constraints on urgent edges. So we will need to consider these valuations separately. The operation $Split(\mathcal{C})$ allows to partition a set of sets of valuations (or a set of zones) into a set of *disjoint* sets of valuations (or a set of disjoint zones). Formally, let $\mathcal{C}$ be a finite set of zones (or constraints). $Split(\mathcal{C})$ is a finite set of zones $\{Z^1, \ldots, Z^n\}$ such that: it partitions the set $\mathcal{C}$ meaning that $\bigcup_{i=1..n} Z^i = \bigcup_{Z \in \mathcal{C}} Z$ and $\forall i \neq j\ Z^i \cap Z^j = \emptyset$; and secondly, its elements "match" the clock constraints of $\mathcal{C}$, meaning that $\forall i \in \{1, \ldots, n\}, \forall Z \in \mathcal{C}, Z^i \subseteq Z$ or $Z^i \cap Z = \emptyset$. For example, $Split(\{1 \leq t < 4, 1 < t < 2, 3 < t < 4\}$ may return $\{t = 1, 1 < t < 2, 2 \leq t \leq 3, 3 < t < 4\}$.

B. Output-update Predecessors of Symbolic States

Definition 5: The set of output-update predecessors of a symbolic state (ℓ, Z_I, Z_O, Z_X), $Pre_o(\ell, Z_I, Z_O, Z_X)$ is defined by:

$$\begin{aligned}Pre_o(\ell, Z_I, Z_O, Z_X) \quad = \quad & \{(\ell', Z_I', Z_O', Z_X') \mid \exists \ell' \xrightarrow{G,A,\mathcal{X}} \ell\ \wedge \\ & Z_I' = Z_I \cap G_I \\ & Z_0' = [A](G_O[A] \cap Z_O) \cap G_O \\ & Z_X' = [\mathcal{X}](G_X[\mathcal{X}] \cap Z_X) \cap G_X\}\end{aligned}$$

Proposition 1: $(\ell', i', o', x') \in Pre_o(\ell, Z_I, Z_O, Z_X)$ iff there is $(\ell, i, o, x) \in (\ell, Z_I, Z_O, Z_X)$ such that $(\ell', i', o', x') \in Pre_o(\ell, i, o, x)$

Proof:

$\Longrightarrow$ This part of the proof is not difficult.

$\Longleftarrow$ Let $(\ell, i, o, x) \in (\ell, Z_I, Z_O, Z_X)$ then $i \in Z_I$, $o \in Z_O$ and $x \in Z_X$. If $(\ell', i', o', x') \in Pre_o(\ell, i, o, x)$ then $i = i'$, $o = o'[A]$ and $x = x'[\mathcal{X}]$ for some $A \in A(O)$. Moreover, according to the semantics of VDTA there exists an edge $\ell' \xrightarrow{G,A,\mathcal{X}} \ell$ such that $i' \in G_I$, $o' \in G_O$ $x' \in G_X$ and additionally $i = i'$, $o = o'[A]$ and $x = x'[\mathcal{X}]$.

1) we get that $i' \in Z_I \cap G_I$ since $i = i'$, $i' \in G_I$ and $i \in Z_I$.
2) Let us show that $o' \in [A](G_O[A] \cap Z_O) \cap G_O$. Since $o = o'[A]$ and $o' \in Proj_O(G)$ we get that $o \in G_O[A]$. Then $o \in G_O[A] \cap Z_O$ since o also belongs to Z_O. Additionally, $o = o'[A]$ implies $o' \in [A]o$. As $o \in G_O[A] \cap Z_O$ we get that $o' \in [A](G_O[A] \cap Z_O)$. Since it is required that $o' \in G_O$ we finally get that

$o' \in [A](G_O[A] \cap Z_O) \cap G_O$.

3) A similar justification is used to show that $x' \in [\mathcal{X}](G_X[\mathcal{X}] \cap Z_X) \cap G_X$.

As there exists an edge $\ell' \xrightarrow{G,A,\mathcal{X}} \ell$, $i' \in Z_I \cap G_I$, $o' \in [A](G_O[A] \cap Z_O) \cap G_O$, and $x' \in [\mathcal{X}](G_X[\mathcal{X}] \cap Z_X) \cap G_X$ we get that $(\ell', i', o', x') \in Pre_o(\ell, Z_I, Z_O, Z_X)$. This ends the proof of the proposition.

C. Input-update Predecessors of Symbolic States

Let us consider the VDTA in Figure 2 and the symbolic state $s_e = (\ell_0, L = 0, true, 1 \le t < 4)$. Assume that one wants to compute $Pre_i(s_e)$. As input update transitions keep the outputs and the clocks unchanged, a simple implementation of Pre_i could only replace in s_e the input zone $L = 0$ by the input zone $true$ (i.e, any value of L) and return $(\ell_0, true, true, 1 \le t < 4)$. Such a simple implementation is not correct since it is not possible to execute an input update transition (the one that sets L to 0) from $(\ell_0, 0, 1, 1.5) \in (\ell_0, true, true, 1 \le t < 4)$ in order to reach s_e. Indeed $(\ell_0, 0, 1, 1.5)$ is not stable and the edge $\ell_0 \xrightarrow{L=0; 1<t<2; S:=1} \ell_1$ is urgently taken. A correct implementation returns stable symbolic states only. On the other hand, a correct implementation may return the following input predecessors for s_e: $(\ell_0, true, true, t = 1), (\ell_0, L \neq 0, true, 1 < t < 2), (\ell_0, true, true, 2 \le t \le 3), (\ell_0, L \neq 0, true, 3 < t < 4)$. Clearly, such an implementation has decomposed the output and clock zones of s_e in order to allow/forbid some input values. The decomposition considers the constraints on the outgoing edge of ℓ_0.

Given a location ℓ, an output and a clock zone Z_O and Z_X we consider the following sets.

The set of constraints on the outgoing edges of ℓ that may become true upon an input update is $Gds(\ell, Z_O, Z_X) = \{G \mid \exists \ell \xrightarrow{G,A,\mathcal{X}} \ell' : G_O \cap Z_O \neq \emptyset$ and $G_X \cap Z_X \neq \emptyset\}$.

The timing context of the tuple (ℓ, Z_O, Z_X) is the set $Ctx_\delta(\ell, Z_O, Z_X) = \{Z_X\} \cup \{G_X \mid G \in Gds(\ell, Z_O, Z_X)\}$.

The output context of the tuple (ℓ, Z_O, Z_X) is the set $Ctx_O(\ell, Z_O, Z_X) = \{Z_O\} \cup \{G_O \mid G \in Gds(\ell, Z_O, Z_X)\}$.

Following the example we get that : $Gds(s_e) = \{(L = 1 \wedge 1 < t < 2), (L = 0 \wedge 3 < t < 4)\}$, $Ctx_\delta(\ell_0, true, 1 \le t < 4) = \{1 \le t < 4, 1 < t < 2, 3 < t < 4\}$ and $Ctx_O(\ell_0, true, 1 < t < 2) = \{true\}$. The computation of $Pre_i(s_e)$ works as follows: We partition the timing context $Ctx_\delta(\ell_0, true, 1 \le t < 4)$ into an equivalent set of disjoint "atomic" clock constraints $\{t = 1, 1 < t < 2, 2 \le t \le 3, 3 < t < 4\}$. Then, for each atomic clock zone Z'_X in set of disjoint "atomic" clock constraints, we partition $Ctx_O(\ell_0, true, Z'_X)$ into an equivalent set of disjoint "atomic" output constraints. In this example, the result is always $\{true\}$. Finally for each "atomic" clock constraints Z'_X and for each atomic output constraint Z'_O the input predecessor compute the negation of input part of constraints in $Gds(\ell, Z'_O, Z'_X)$.

Let us abstract the above thought into a formal definition.

Definition 6: The set of input-update predecessors of a symbolic state (ℓ, Z_I, Z_O, Z_X), $Pre_i(\ell, Z_I, Z_O, Z_X)$ is defined by:

$$\begin{aligned} Pre_i(\ell, Z_I, Z_O, Z_X) \quad = \quad & \{(\ell', Z'_I, Z'_O, Z'_X) \mid \\ & \ell' = \ell, \\ & Z'_X \in Split(Ctx_\delta(\ell, Z_O, Z_X)), \\ & Z'_O \in Split(Ctx_O(\ell, Z_O, Z'_X)), \\ & Z'_I \in \bigcap_{G' \in Gds(\ell, G'_O, Z'_X)} \neg G'_I\} \end{aligned}$$

One can show the following.

Proposition 2: $(\ell', i', o', x') \in Pre_i(\ell, Z_I, Z_O, Z_X)$ iff there is $(\ell, i, o, x) \in (\ell, Z_I, Z_O, Z_X)$ such that $(\ell', i', o', x') \in Pre_i(\ell, i, o, x)$

D. Time Predecessors of Symbolic States

The time predecessor $Pre_t(\ell, Z_I, Z_O, Z_X)$ is implemented analogously to Pre_i. Similarly to input updates, the time elapses in stable states only and then we also need to carefully decompose constraints. As far as time elapsing transitions only change the value of clocks, we decompose the input and the output constraints.

We define the sets $Gds(\ell, Z_I, Z_O) = \{G \mid \exists \ell \xrightarrow{G,A,\mathcal{X}} \ell' : G_I \cap Z_I \neq \emptyset$ and $G_O \cap Z_O \neq \emptyset\}$, $Ctx_I(\ell, Z_I, Z_O) = \{Z_I\} \cup \{G_I \mid G \in Gds(\ell, Z_I, Z_O)\}$ and $Ctx_O(\ell, Z_I, Z_O) = \{Z_O\} \cup \{G_O \mid G \in Gds(\ell, Z_I, Z_O)\}$. The set $Gds(\ell, Z_I, Z_O)$ returns the constraints on the outgoing edges of ℓ that may become true when one looks back into the time from (ℓ, Z_I, Z_O, Z_X).

Consider Figure 2 and assume that one wants to compute $Pre_t(s_2)$ where $s_2 = (\ell_0, L = 0, true, 3 < t < 4)$. Note that although $0 \le t < 4$ is the past of $3 < t < 4$, the symbolic state $(\ell_0, L = 0, true, 0 \le t < 4)$ is not a time predecessor of s_t since it is not possible to reach s_2 from $(\ell_0, 0, 0, 1.3) \in (\ell_0, L = 0, true, 0 \le t < 4)$ by letting the time elapse. Indeed, $(\ell_0, 0, 0, 1.3)$ is not stable. Then, we consider $Gds(L = 0, true) = \{L = 0 \wedge 3 < t < 4, L = 0 \wedge 1 < t < 2\}$, $Ctx_I(L = 0, true) = \{L = 0\}$ and $Ctx_O(L = 0, true) = \{true\}$. In a first time one decomposes $Ctx_I(L = 0, true)$ into a set of disjoint "atomic" input constraints and one gets $\{L = 0\}$. Secondly one decomposes $Ctx_O(L = 0, true)$ into a set of disjoint "atomic" output constraints and one gets $\{true\}$. The unique input/output context obtained after the decompositions is given by $L = 0$ and $true$. Then, from the symbolic state $(\ell_0, L = 0, true, 3 < t < 4)$ we compute a clock zone Z' from which we can reach $3 < t < 4$ by time elapsing and along the (time) path from Z' to $3 < t < 4$ we must avoid the clock part of all constraints in $Gds(L = 0, true)$ since otherwise one reach non stable states. This correspond to the computation of *safe time predecessors* of a clock zone with respect to another one. For instance we compute the safe time predecessor of $3 < t < 4$ with respect to $\{3 < t < 4, 1 < t < 2\}$ and we should get $2 \leq t \leq 3$. Finally we should get that $Pre_t(s_2) = (\ell_0, L = 0, true, 2 \leq t \leq 3)$.

For the computation of $Pre_t(\ell, Z_I, Z_O, Z_X)$, the next step after the decomposition of input/output constraints is to compute the *safe time predecessors*. We define the time predecessor as follows:

Definition 7: The set of time predecessors of a symbolic state (ℓ, Z_I, Z_O, Z_X), $Pre_i(\ell, Z_I, Z_O, Z_X)$ is defined by:

$$\begin{aligned} Pre_t(\ell, Z_I, Z_O, Z_X) = \{(\ell', Z'_I, Z'_O, Z'_X) \mid \\ \ell' = \ell, \\ Z'_I \in Split(Ctx_I(\ell, Z_I, Z_O)), \\ Z'_O \in Split(Ctx_O(\ell, Z'_I, Z_O)), \\ Z'_X \in Pre^s_t(Z_X, \{G_X \mid G \in Gds(\ell, Z'_I, Z'_O)\})\} \end{aligned}$$

One can show the following.

Proposition 3: $(\ell', i', o', x') \in Pre_t(\ell, Z_I, Z_O, Z_X)$ iff there is $(\ell, i, o, x) \in (\ell, Z_I, Z_O, Z_X)$ such that $(\ell', i', o', x') \in Pre_t(\ell, i, o, x)$

E. Zone-Based Reachability Analysis Algorithm

The zone based reachability analysis algorithm computes at each step the predecessors of already encountered symbolic states. The predecessors are computed as follows:

$$Pre(\ell, Z_I, Z_O, Z_X) = \bigcup_{\tau \in \{i,o,t\}} Pre_\tau(\ell, Z_I, Z_O, Z_X)$$

Note that all the operations on zones (update, pre-update, past time, $Split$) can be effectively computed [4] using DBM (or simple adaptations for input and output zones). The iterative fixpoint process of Algorithm 1 converges after finite many steps. This is true as the computation of Pre_i, Pre_o, Pre^s_t and Pre_t involves exactly the constants in $\mathcal{A}$ and variable updates only set variables to constants. Using similar argument to [6], one can show that the reachability analysis algorithm is EXPTIME. Although we did not carry out experiments of the zone-based algorithm, it should be clear that this new algorithm for VDTA is practically more efficient than that based on regions [11].

IV. Conclusion and Future Work

The paper proposed a more efficient zone-based backward algorithm for the reachability analysis of Variable Driven Timed Automata (VDTA). The algorithm is based on the definition of predecessors operators of symbolic states. Symbolic states contain input, output, and clock constraints that should be respected in order to reach the target state. These informations can be used by the environment to control the execution of systems or select test cases.

The VDTA model received a favorable echo among some industrial partners since it is a good compromise between simplicity and expressiveness. Ongoing works include composability issues. Moreover, one can intensify the model with tough arithmetic operations and study decidability issues.

Acknowledgment

This work has been supported by the ANR (Agence Nationale de la Recherche) TesTEC Project. We also would like to thank Hervé Marchand for his comments.

References

[1] Rajeev Alur, Thao Dang, and Franjo Ivancic. Reachability analysis of hybrid systems via predicate abstraction. In *5th International Workshop on Hybrid Systems: Computation and Control (HSCC'02)*, pages 35–48. Springer, 2002.

[2] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] Roberto Barbuti and Luca Tesei. Timed automata with urgent transitions. *Acta Informatica*, 40(5), 2004.

[4] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithm and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124. Springer, 2004.

[5] Patricia Bouyer. Untameable timed automata! In *20th Symposium on Theoretical Aspects of Computer Science (STACS'03)*, pages 620–631. Springer, 2003.

[6] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *16th International Conference on Concurrency Theory (CONCUR'05)*, pages 66–80. Springer, 2005.

[7] Frédéric Herbreteau and B. Srivathsan. Efficient on-the-fly emptiness check for timed büchi automata. In *8th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'10)*, pages 218–232. Springer, 2010.

[8] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed i/o automata: A mathematical framework for modeling and analyzing real-time systems. In *The 24th IEEE International Real-Time Systems Symposium (RTSS'03)*, volume 0, pages 166–177. IEEE, 2003.

[9] Stephan Merz. Model checking: A tutorial overview. In *Modeling and Verification of Parallel Processes*, pages 3–38. Springer, 2001.

[10] Houda Bel Mokadem, Béatrice Bérard, Patricia Bouyer, and François Laroussinie. A new modality for almost everywhere properties in timed automata. In *16th International Conference on Concurrency Theory (CONCUR'05)*, pages 110–124. Springer, 2005.

[11] Omer Nguena-Timo, Hervé Marchand, and Antoine Rollet. Automatic test generation for data-flow reactive systems with time constraints. In *22nd IFIP International Conference on Testing Software and Systems: Short Papers (ICTSS'10)*. CRIM-Canada, 2010.

[12] Omer Nguena-Timo and Antoine Rollet. Conformance testing of variable driven automata. In *8th IEEE International Workshop on Factory Communication Systems Communication in Automation (WFCS'10)*, 2010.

[13] Omer Nguena-Timo and Antoine Rollet. Test selection for data-flow reactive systems based on observations. In *Software Testing, Verification and Validation Workshops (ICSTW'11)*. IEEE Xplore, 2011.

[14] Peter J. Ramadge and W. Murray Wohnam. The control of discrete event systems. *Proceedings of the IEEE Computer Society*, 77:81–98, 1989.

[15] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software-Concepts and Tools*, 17:103–120, 1996.

Retrospective Project Analysis Using the Expectation-Maximization Clustering Algorithm

Steffen Herbold, Jens Grabowski, Stephan Waack
Institute of Computer Science
Georg-August-Universität Göttingen, Germany
Email: {herbold, grabowski, waack}@cs.uni-goettingen.de

***Abstract*—Schedule slips are often the reason for failed projects or low-quality software. Therefore, investigation if a project was on schedule is an important task when analyzing software projects in retrospective. In this paper, we present a data-driven approach for the retrospective determination of project phases through a clustering algorithm. The analysis is based on software metrics measured at different points of time during the project execution. We will describe how the data can be collected, prepared and analyzed. Our findings are validated through a case study where we analyzed two large-scale open-source projects. The results show that it is possible to successfully identify the final phase of a project using our approach.**

***Keywords*-EM clustering, project analysis, repository mining**

I. Introduction

One of the biggest challenges of software projects other than the task at hand is the project plan. Often, time seems too short and schedule slips occur, or features have to be removed to remain on schedule with milestones and release candidates. On the other hand, this can also cause developers to ignore parts of the schedule, e.g., they add features after a feature freeze, instead of focussing on stabilizing the project. This increases the risk of producing low-quality software, which will reduce costumer satisfaction and increase the maintainance costs. It is the task of good software development processes to prevent this. To improve the current process, the retrospective analysis of past projects with respect to their schedule is an important means. This is often performed using experts intuition, based on tangible data consciously and unconsciously taken into account. This can include knowledge about the project environment as well as information about the project itself, e.g., the size or the number of unresolved bugs.

In our previous work [1], we have successfully used the k-means algorithm [2] to identify feature freezes during a project. The approach is based on software metric data about the project's milestones mined from the projects repository. The contribution of this paper is an extension of this approach. First, we extend the points of time for the measurement from milestones to arbitrary points of time $t_1, \dots, t_n$, e.g., the milestones but also weekly measurements. This allows steering of the coarseness of the analysis. For the clustering, we propose the Expectation-Maximization (EM) clustering algorithm [3], because it is more powerful than k-means. The assumption of the approach is that metric values are similar if they are measured during the same phase of a project and different, if they are from different phases. For that reason, we use clusters as indicators about the actual project phases and when they changed.

The remainder of this paper is structured as follows. In Section II, we introduce software metrics and discuss which metrics we use and how we selected them. Afterwards, in Section III, we explain the basic concept of the EM clustering algorithm. Section IV discusses our methodology for data collection, preparation, and analysis. In Section V, we apply our approach in a case study. The results of our work and the threats to its validity are discussed in VI. Finally, Section VII concludes the paper and gives an outlook on future work on this subject.

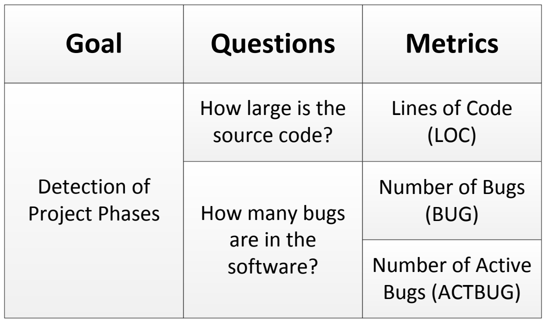
II. Software Metrics

The first problem when analyzing software or software projects is that software is an abstract and difficult to grasp product. Software metrics are a means to describe the abstract product software with numbers. The IEEE defines software metrics as “the quantitative measure of the degree to which a system, component, or process possesses a given attribute” [4]. For our analysis, we need quantifiable attributes of software development projects, which is exactly what software metrics can provide.

We use a target-oriented approach for the selection of software metrics, the *Goal/Question/Metric* (GQM) approach [5], [6]. In this approach, first a *goal* that shall be achieved is defined. Then, *questions* are formulated whose answer can be used to achieve the goal. Finally, *metrics* that can answer the questions are selected. This methodology ensures that there is no ‘measurement for the sake of measurement’, but that it is clear why the metric data is collected and how it is used.

We applied the GQM approach to select metrics to achieve our goal, the *detection of project phases* (Figure 1). We defined two questions to evaluate the goal.

1) How large is the source code?
2) How many bugs are in the software?



Goal	Questions	Metrics
Detection of Project Phases	How large is the source code?	Lines of Code (LOC)
	How many bugs are in the software?	Number of Bugs (BUG)
		Number of Active Bugs (ACTBUG)

Figure 1. GQM approach to select apropriate metrics.

The rational behind these questions is that we feel that the most important features to determine the progress of a project are the software's size and its number of bugs. The size determines how much of the software's source code is written and should increase continously as the projects progresses. The number of bugs is an indicator for the stability of a project. It should drop sharply at the end of the project, as the focus switches from development to stabilizing the project. For the first question, we selected the metric *Lines of Code* (LOC) to measure the size of the software. For the second question, we determined two similar candidates: *Number of Bugs* (BUG) and *Number of Active Bugs* (ACTBUG). The metric BUG measures how many of the total number of bugs known about the software at the end of the project are still open, i.e., have not yet been fixed. The metric ACTBUG includes when a bug has been reported, i.e., it does not measure the number of open bugs with relation to the end of the project, but to the currently known bugs. To our mind, one of these two metrics should be sufficient, because it can be shown that a small number of metrics often performs similar to larger sets [7]. As part of the case studies we plan to investigate this further and evaluate if either BUG or ACTBUG performs better than the other.

III. The EM algorithm

To analyze the data, we use the *EM clustering algorithm* [3], which belongs to the *unsupervised* learning algorithms. Unsupervised means that no prior knowledge except the data is used. In our case, this means that the algorithm does not know when a data point has been measured and which project phase it belongs to, only the metric data itself is known. Clustering algorithms estimate the data sources that created the data. In our case, the data sources are project phases. The EM algorithm determines a mixture of gaussian distributions that fits the data. This mixture is basically a number of k gaussian distributions and each data point "belongs" to the distribution which generated it with the highest probability. Each of the distribitions defines a *cluster*, i.e., a set of points that the algorithm determines to be generated by the same data source. The points are assigned to the clusters based on the likelyhood that the underlying gaussian distribution generated the data point. The number k of clusters is not fixed, but determined by the algorithm itself.

The acronym EM stands for *expectation maximization* and describes the two basic steps of the algorithm: 1) calculate the *expected* likelihood of the current hypothesis; 2) determine a new hypothesis to *maximize* the likelihood. Additional details of the algorithm can be found in [8].

IV. Approach

Our approach for the retrospective analysis of software project consists of three phases: 1) data collection; 2) data preparation; 3) data analysis.

A. Data Collection

We selected the metrics LOC, BUG, and ACTBUG for the evaluation (see Section II). For the analysis, we need the values of these metrics at regular points of time $t_1, \ldots, t_n$ during the project. To collect the metric data in retrospective, access to the software project's repository is necessary.

Source code based metrics, like LOC, can be extracted from a *code versioning system*, e.g., *Concurrent Versions System* (CVS) [9], *Subversion* (SVN) [10] or Git [11]. These system allow the access to the whole history of the source code. That way, the state of source code at the times $t_1, \ldots, t_n$ can be accessed. Once the source code is available, we can measure the LOC with any software measurement tool.

The metrics BUG and ACTBUG are gathered from *bug-tracking systems*, e.g., Bugzilla [12]. Bugtracking systems maintain all data related to bugs, i.e., when they were discovered, in which versions of the software they are present, the current state of the bug, a record or its state-changes, and how it was resolved. Possible states of the bugs are, e.g., OPEN and CLOSED: OPEN indicates that a bug is reported and still being worked on; CLOSED means that the bug is resolved. Possible resolutions are, e.g., FIXED, INVALID, and WONTFIX: FIXED means that a bug has been corrected; INVALID means that the entry is not a bug; WONTFIX means that for some reason the bug will not be fixed. Using all these informations, we extract all known bugs for a specific version of a software to measure the metric BUG. By including the information when a bug has been reported, we measure the metric ACTBUG.

The result of the data collection are the metric values for LOC, BUG, and ACTBUG at times $t_1, \ldots, t_n$.

B. Data Preparation

The collected data needs to be prepared for the analysis. To this aim, we *normalize* the data. Normalization means, that we change the scales of the metrics to the interval $[0, 1]$ while keeping the relative distances between the metric values. The normalized metric values $value_{norm}$

are calculated as $value_{norm} = \frac{value - value_{min}}{value_{max} - value_{min}}$, where $value_{min}, value_{max}$ represent the minimal and maximal measured values of the metric.

The reason for the normalization is to reduce the impact of the metric scales. The different scales of the metrics effect the result of the clustering. The scale of LOC is much larger than the scale of BUG and ACTBUG. This difference can give LOC a higher weight than the other two metrics. The important feature for the analysis are not the absolute values but the relative distances to the other values in the project, because the relative distance reflect the progress. Normalization keeps the relative distances, but removes the scale effects, thereby allowing a better data analysis.

As result of the data preparation, we have a the data set *DATA* $= \{values(t_1), \dots, values(t_n)\} \subset [0,1]^3$. The notation $values(t_i)$ stands for the value of metrics LOC, BUG and ACTBUG at t_i.

C. Data Analysis

For the data analysis, we use the EM clustering algorithm and apply it to the metric data. The input of the algorithm is only the metric data itself and no information about project phases according to the project plan or even the date of the measured. As result, the algorithm yields k clusters $C_1, \dots, C_k \subset$ *DATA*. The clusters are a disjoint partition of the input, i.e., $\bigcup_{i=1}^{k} C_i =$ *DATA* and $C_i \cap C_j = \emptyset$ for all $i, j = 1, \dots, k$. The number k is not fixed and the algorithm can determine as many or few clusters as required to fit the data.

If the analysis is successfull, the resulting clusters contain time-adjacent data, i.e., $C_i = \{values(t_j), values(t_j + 1), \dots, values(t_{j+|C_i|})\}$. Such a cluster describes a time-interval $[t_j, t_{j+|C_i|}]$. The time-intervals can then be mapped to the project phases by an expert to gain knowledge about the project. In case the resulting clusters are not time-adjacent, there are to possible conclusions: 1) the approach failed; 2) the project was chaotic. Which of the two is the case needs to be determined by an expert.

V. Case Studies

To validate our approach, we performed a case study where we applied it to two large-scale open source projects. We designed our case study to answer the following two research questions:

- RQ1: Is the approach able to identify project phases?
- RQ2: is either BUG or ACTBUG sufficient or are both required?

In the following, we will describe the case study methodology and data. Then, we will present the results of the experiments. Based on the results, we answer the research questions in Section VI.

A. Methodology and Data

The experiments we performed in this case study are based on data obtained from the development of projects hosted by the Eclipse Foundation [13]. We mined data about the developement of two versions of the Eclipse Platform project [14], the versions 3.2 and 3.3. We excluded the *Standard Widget Toolkit* (SWT) subproject of the platform from our measurements, as it is for the most part independent of the remainder of the project. We obtained the source code from the Eclipse CVS repository [15]. To measure the bug related metrics, we used SQL queries to directly extract the metrics from a database dump of the Eclipse Bugzilla [16] bugtracking system made available to us.

As dates for the measurements we choose weekly intervals on monday mornings. For the Eclipse Platform 3.2 the starting date was 2005-06-27 and the final measurement was 2006-06-26. For the Eclipse Platform 3.3 the start date was 2006-06-26 and the final date 2007-06-26. For the analysis we used Weka's [17] implementation of the EM clustering algorithm, with a maximum of 100 iterations and a dynamic number of clusters.

To answer the research question, we performed three experiments with both software versions respectively. In the first experiment, we used all three metrics as input for the EM clustering algoritm, in the second experiment we used only the metrics LOC and BUG, and in the third experiment we used only the metrics LOC and ACTBUG. The first experiment is to evaluate if the identification of project phases through the clustering works to answer RQ1. The other two experiments evaluate how each of the bug metrics alone performs in order to answer RQ2.

B. Results

The results of the experiments are visualized in figures 2-7. The figures depict the weekly measured data points for the metric used in the experiments and where the clusters are located. Both projects had several milestones and release candidates. MX stands for milestone X, M0 denotes the beginning of the project. RCX denote release candidate X. The number of clusters varies between two and four, depending on the experiment. Each cluster only contains time-adjacent data. While the number of clusters varies, the last cluster found is in all six experiments very similar. The cluster is within 3 weeks of the first release candidate. The clusters before the last one are inconsistent and vary.

VI. Discussion

In this section we discuss the case study results and use them to answer our research questions. Furthermore, we list the threats to the validity of our studies.

A. RQ1: Is the approach able to identify project phases?

All clusters determined in the case study were time-adjacent, thereby providing evidence that cluster analysis is

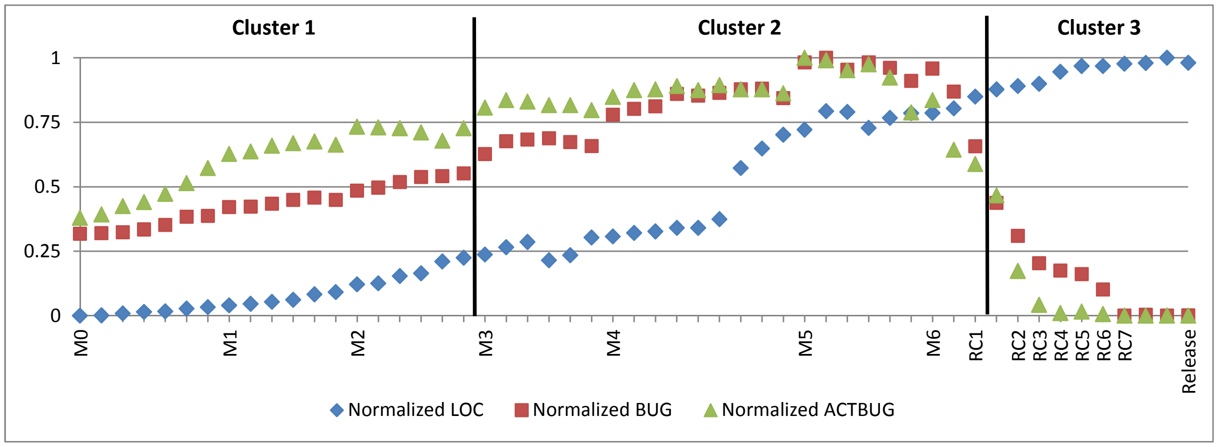


Figure 2. EM clustering with normalized LOC, BUG, and ACTBUG for the Eclipse Platform 3.2

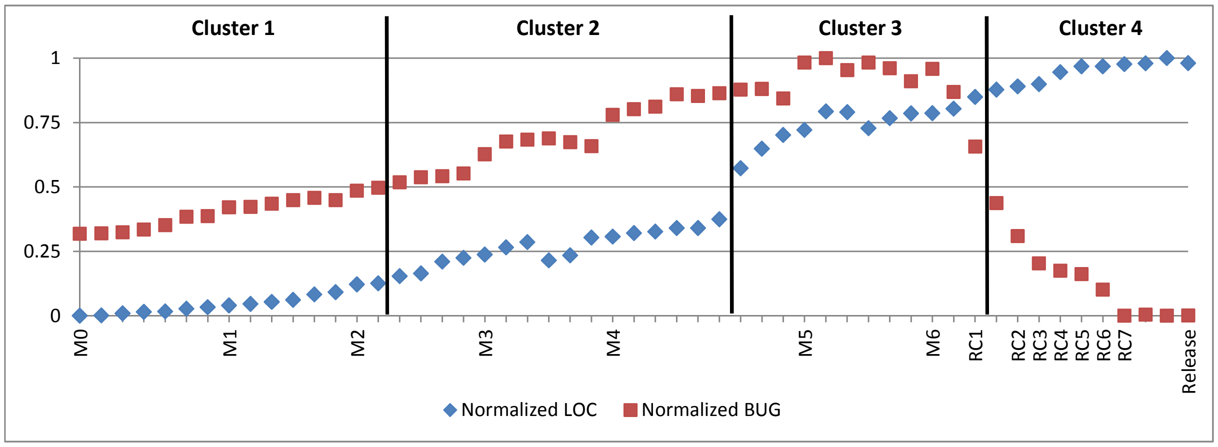


Figure 3. EM clustering with normalized LOC and BUG for the Eclipse Platform 3.2

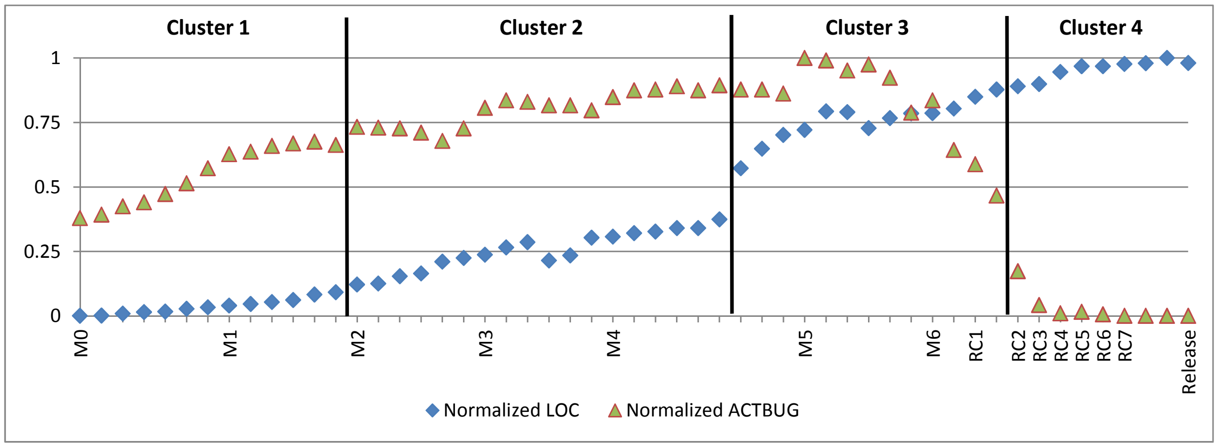


Figure 4. EM clustering with normalized LOC and ACTBUG for the Eclipse Platform 3.2

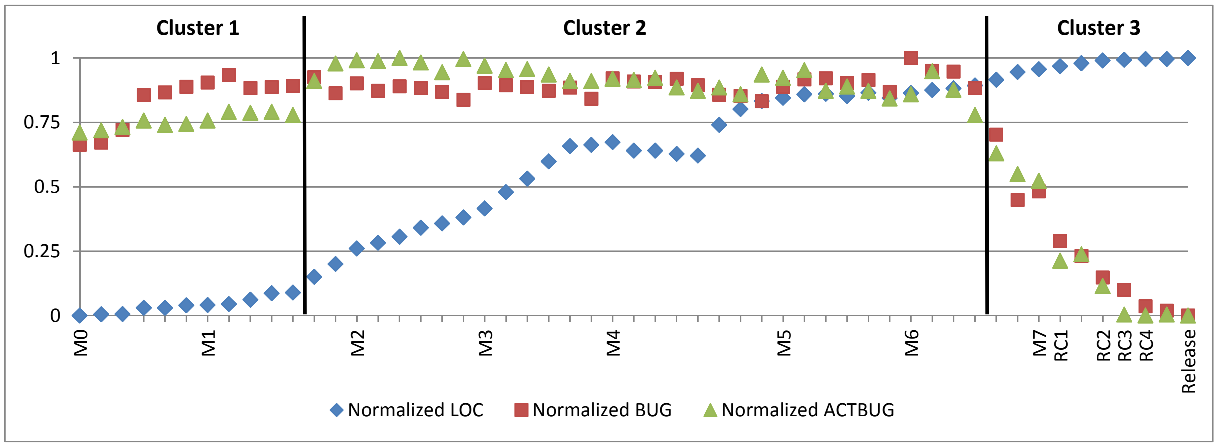


Figure 5. EM clustering with normalized LOC, BUG, and ACTBUG for the Eclipse Platform 3.3

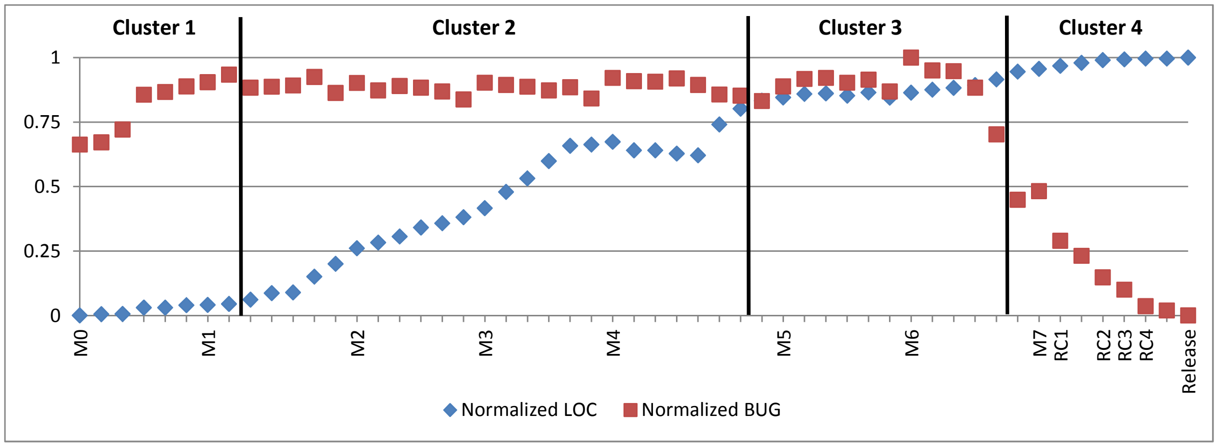


Figure 6. EM clustering with normalized LOC and BUG for the Eclipse Platform 3.3

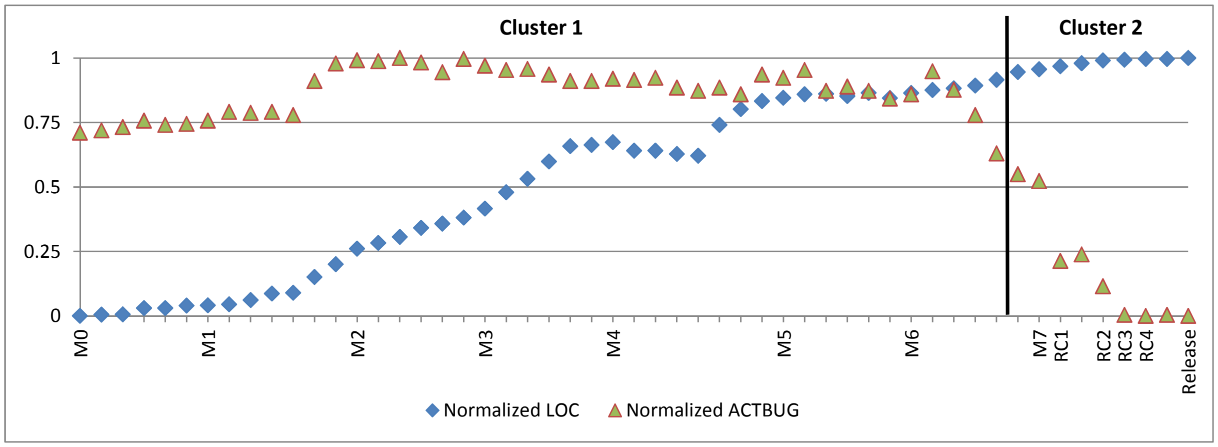


Figure 7. EM clustering with normalized LOC and ACTBUG for the Eclipse Platform 3.3

indead capable of partitioning a project into phases. The case study results show that the final phase of the project leading up to the release can be detected with good accuracy. With regard to this phase, we answer this research question with yes. However, other phases could not accurately be determined and the results varied between the experiments. For example, with the metrics LOC and BUG and for the Eclipse Platform 3.3 (Figure 6), the first cluster seems to be an initializing project phase, where the project was still in planning mode. Thus, this experiment did not only detect the final phase, but also the intial one. But the detection is singular and not repeated accurately in other experiments. Therefore, the approach seems to be capable of detecting further phases, but is not reliable.

B. RQ2: Is either BUG or ACTBUG sufficient or are both required?

When it comes to detecting the final phase of a project, the results show no significant difference between using both BUG and ACTBUG or only one of them. Thus, any of the three combinations is feasible. Therefore, one should use either only BUG or ACTBUG instead of using both, as it reduces the demands on the data mining as well as the dimension of the data set, thereby simplifying the analysis.

C. Threats to validity

There are several threats to the validity of our results. Our case study was only performed for successful industrial open-source projects. We did not consider closed-source projects, or community driven open-source projects. Furthermore, both projects are consecutive version of the same software. The results of the case study are only consistent when it comes to the detection of the final project phase and inconsistent otherwise, indicating possible problems with the analysis.

VII. Conclusion

In this paper, we defined an approach for the retrospective analysis of software development projects. The approach is purely data-driven and based on software metrics. We described how we selected appropriate metrics using the GQM approach. As basis for the analysis we use metric data measured at different times during the execution of a project. We then partition the data into clusters using the EM clustering algorithm. Aim of the analysis is to map the clusters to phases of the project. Our case studies showed that the approach can accurately determine the final phase of a project, but has problem detecting prior phases.

Future work on this project has several promising directions. First, it is possible to tweak the clustering algorithm used for the analysis, e.g., by predefining a number of clusters that matches the project plan. A detailed comparison with other clustering algorithms should also be explored. Second, the metric set can be extended with further metrics. For example, the number of successfull tests or the overall complexity of the project. Third, the time intervals used for the measurement can also be varied to try to determine whether they have an effect on the results. Finally, we will consider further projects to broaden the scope of the case studies.

References

[1] S. Herbold, J. Grabowski, H. Neukirchen, and S. Waack, “Retrospective Analysis of Software Projects using k-Means Clustering,” in *Proc. of the 2nd Design for Future 2010 Workshop (DFF 2010), Bad Honnef, Germany*, May 2010.

[2] D. J. MacKay, *Information theory, inference, and learning algorithms*. Cambridge Univ. Press, 2003.

[3] A. P. Dempster, N. M. Laird, and D. B. Rubin, “Maximum Likelihood from Incomplete Data via the EM Algorithm,” *J. Royal Statistical Soc.*, vol. 39, no. 1, pp. 1–38, 1977. [Online]. Available: http://www.jstor.org/stable/2984875

[4] IEEE, “Ieee glossary of software engineering terminology, ieee standard 610.12,” IEEE, Tech. Rep., 1990.

[5] V. Basili and D. Weiss, “A methodology for collecting valid software engineering data,” *IEEE Trans. Softw. Eng.*, vol. 10, no. 6, pp. 728–738, 1984.

[6] V. Basili and H. Rombach, “The TAME project: towards improvement-oriented softwareenvironments,” *IEEE Trans. Softw. Eng.*, vol. 14, no. 6, pp. 758–773, 1988.

[7] S. Herbold, J. Grabowski, and S. Waack, “Calculation and Optimization of Thresholds for Sets of Software Metrics,” *Empirical Softw. Eng.*, pp. 1–30, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9162-z

[8] T. Mitchell, *Machine Learning (Mcgraw-Hill International Edit)*, 1st ed. McGraw-Hill Education (ISE Editions), Oct. 1997. [Online]. Available: http://www.worldcat.org/isbn/0071154671

[9] July 2011. [Online]. Available: http://www.nongnu.org/cvs/

[10] July 2011. [Online]. Available: http://subversion.apache.org/

[11] July 2011. [Online]. Available: http://git-scm.com/

[12] July 2011. [Online]. Available: http://www.bugzilla.org/

[13] July 2011. [Online]. Available: http://www.eclipse.org/

[14] July 2011. [Online]. Available: http://www.eclipse.org/platform/

[15] July 2011. [Online]. Available: dev.eclipse.org:/cvsroot/eclipse

[16] July 2011. [Online]. Available: https://bugs.eclipse.org/bugs/

[17] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The weka data mining software: an update,” *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

Extracting and Verifying Viewpoints Models in Multitask Applications

Selma Azaiez, Belgacem Ben Hedia, Vincent David
CEA, LIST, Embedded Real Time Systems Laboratory,
Point Courrier 94, Gif-sur-Yvette, F-91191 France
Email: {selma.azaiez,belgacem.ben-hedia,vincent.david}@cea.fr

Abstract—**Static analyzers are most of the time dedicated to checking runtime errors in sequential programs or are specific to one particular property in the multitasking domain such as deadlock detection. However, the safety of multitask and realtime applications relies on several properties (e.g., absence of deadlock, atomicity, respecting the temporal constraints, etc.). Verification of each property requires a specific abstract model. In this paper, we introduce a generic pattern-based method allowing automatic extraction of viewpoint models appropriate for verification of various properties. Each property is defined by a property analysis pattern specifying the algorithm for its verification (steps are defined to specify needed viewpoint models). The extraction of each viewpoint model is described within a dedicated model extraction pattern. Property analysis patterns and model extraction ones are the main achievement of our work. By introducing these patterns, our method allows harmonizing the validation process and capitalizing the knowhow by explicitly defining the verification and transformation processes.**

Keywords- multitask applications; semantic-based static analysis; property verification; property analysis pattern; model extraction pattern.

I. INTRODUCTION

To validate multitask and real-time systems, developers or validators in independent certification authorities have to use different tools based on different methods (e.g., static analysis [5] or model checking [3]). Each tool is dedicated to a specific class of properties or even to a specific stage in the development process [15][16]. For instance, model checking tools are usually used to validate system specifications (expressed using dedicated formal languages); which does not really reflect what is really implemented. In the other hand, the safety of multitask and real-time applications relies on the satisfaction of several properties such as deadlock freedom, atomicity, respecting the temporal constraints, etc. The verification of each property is based on a different viewpoint model. For instance, models focusing on locks are necessary to detect deadlocks; models focusing on shared memory – to check atomicity. Using such different techniques and tools complicate the validation process and a high expertise is required for each type of property.

In this paper, we propose an approach targeting to harmonize validation process for multitask and realtime systems. The approach is applicable to check correctness in these systems from their source code. It is based on the extraction of different viewpoint models driven by the properties to verify. It uses property analysis and model extraction patterns. A property analysis pattern defines the algorithm for determining which models have to be extracted in order to verify the property. The extraction process of each model is described within a model extraction pattern.

The paper is structured as follows. In Section II, we provide an overview of our approach. In Section III, we explain the semantic annotation. Then, in Sections IV and V, we introduce property analysis and model extraction patterns, respectively. Finally, in Section VI we illustrate our approach on a simple example.

II. RELATED WORK AND APPROACH OVERVIEW

To address our topic, we study different types of validation and verification techniques. We first look at techniques dealing with source code analysis: static analysis [5] and reverse engineering [19]. Then, we study model checking and theorem proving techniques that are suitable for verifying multitasking and realtime systems.

Most static analysis tools were developed for detecting numerical software bugs in sequential programs [15] (e.g., buffer overflows or underflows, null pointer references, etc.). Some examples of such analyzers are ASTREE [10], CAVEAT [11], PolySpace[12], Coverity [13], etc. Other existing tools are more suitable for our context (i.e., multitask realtime systems) but are specific to particular types of properties (e.g., deadlock freedom or race condition detection) [14][17][18]. In reverse engineering approaches, some tools [20] are only focusing on generating structural models such as UML diagram class or function dependencies while others are based on model checking techniques [21][22] but do not address concurrency issues. Hence, in both areas limited concurrency and realtime issues are addressed. By contrast, model checking and theorem proving techniques [4][6][7][8][9] focuses on safety properties in multitask realtime systems. However, verifications are performed on systems specifications which make them suitable for top-down approaches where developers need to be sure that their programs are correct by construction. In our context, a bottom-up approach is needed where different viewpoint models can be extracted from the source code driven by the properties to verify.

The method we propose allows bridging the gap between techniques such as static analysis that checks source code and techniques such as model checking and theorem proving that uses more abstract models. It is based on three main levels. In the first level, source code is parsed and the AST (Abstract Syntax Tree) is produced. In the second level, AST nodes are annotated. The annotation allows capturing the semantics of specific real-time multitasking APIs objects (such as those provided by POSIX [1], OSEK VDX[2], etc.). A formalism (introduced in Section III) is used for this purpose. During annotation phase, nodes using multitasking elements (primitives or variables) are identified and they are assigned with annotations provided by the semantic description.

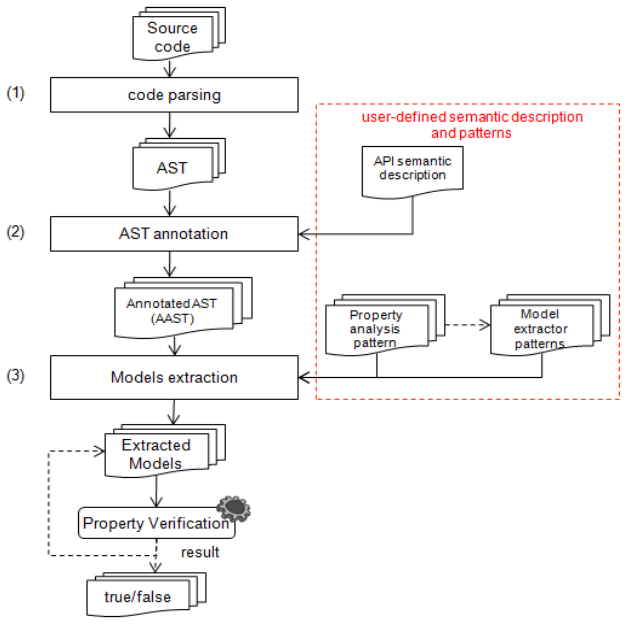


Figure 1. Layers of analysis

In the third level, models are extracted and properties are verified according to user-defined patterns. *Properties analysis patterns* specify within a *checking process* which models are necessary to perform the verification.

Model extraction patterns define *transformation rules* according to *pre-conditions* describing initial model configuration and *post-conditions* describing configuration of output model. Once models are extracted the verification of the property is performed. The result of the verification process can either be a Boolean value or an extracted model.

III. SEMANTIC ANNOTATION

Multitasking concepts are usually implemented within API(s) that include a set of functionalities, data types, data structures, and protocols aiming to facilitate access to resources or services. The use of API(s) elements in the source code is identified during AST annotation phase according to the provided semantic description. In this section, we introduce the formalism that allows capturing API semantics. Then, we introduce features that facilitate models extraction from the AAST (Annotated Abstract Syntax Tree).

A. *Semantic description*

We introduce a set of annotations in order to capture multitask API semantics, based on two main concepts:

- Semantic primitives: functions introduced by the API;
- Semantic variables: constraints on the type and values of the parameters and values returned by the primitive call.

1) *Multitask primitives*

Multitask primitives are classified upon their general semantics:

- Task management: task creation, destruction, sleep and awakening;
- Critical sections management: locks acquisition and release;
- Communication mechanisms: creation or destruction of message-passing mechanisms or shared memories, sending and receiving of messages.

A multitasking primitive is described as follows:

$$P = (id,\ \varphi(parami),\ \varphi(res),\ sem\text{-}role)$$

where:

- `id` : is the identifier of the primitive (e.g., fork);
- $\varphi(param_i)$: value and type constraints having to be respected by primitive parameters;
- $\varphi(res)$: value and type constraints that have to be respected by the primitive return value;
- *`sem-role:`* annotation expressing the primitive role (e.g., *`CREATE-TASK`*, *`TAKE-LOCK`*, *`RELEASE-LOCK`*, *`SLEEP`*, etc.).

2) *Semantic variables*

Semantic variables are the parameters or return values of a primitive. They are defined as following:

$$V = (id,\ \varphi(type),\ range)$$

where:

- `id`: is an identifier (used in semantic description of the primitive);
- $\varphi(type)$: are type constraints;
- *`range`*: is the range of acceptable values or a constant.

B. *AAST node structure*

During parsing phase, the program is tokenized then, the AST is generated. A node in the AST is associated to each word in the source code. Additional nodes are added that we call branch-node and that inform about the syntactical nature of the branch (e.g., statement, function call, loop body, else body etc.). A token is associated to each node to inform about the node nature (e.g., FUNC-CALL, END-LOOP, etc.).

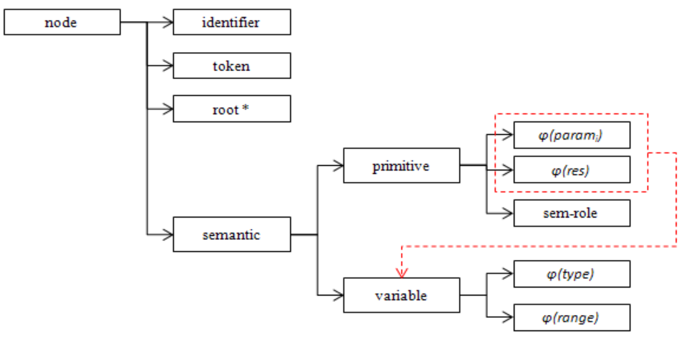


Figure 2. The structure of the annotated AST node

During annotation phase, AST nodes conserve information about syntactical structures. This information is augmented by semantic information mentioned in the previous Sections. After semantic annotation phase, AST nodes have the structure shown in Figure 2.

C. *From AAST to model extraction*

Model extraction consists in applying a set of transformation rules to extract a new model configuration from one or several initial ones. To be able to define such rules, original and target models have to follow a common abstract definition. In this subsection, we introduce abstract definition of models as well as a navigation feature that will facilitate the definition of transformation rules.

1) *Abstract definition of models*

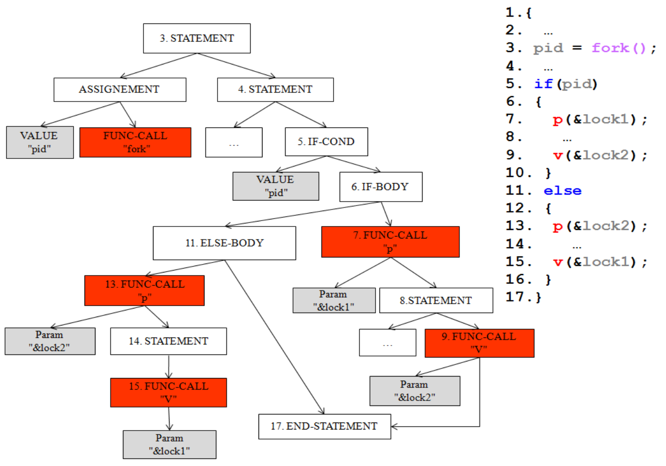
A common representation to all models used in our approach (i.e., AAST, control graphs, synchronization and communication graphs) is graph representation. For all these representation, we adopt a generic definition stating that a graph is a quadruple $G(S,\ T,\ s_0,\ s_f)$ where

- S: is a set of nodes;
- T: is a set of transitions which can be labeled or not;
- $s_0 \subset S$: is the initial node;
- $s_e \subset S$: is the set of final nodes.

2) *Navigation formalism*

During patterns specification, one can need to select specific model elements (nodes or transition sets) or to test nodes according to their identifiers, tokens, branches to which they belong or their semantic annotation. To this purpose, we introduce the operator "`::`" that allows such navigation. For instance, the expression G::S states that we refer to the set of nodes S in the graph G.

In Figure 3, we provide an example written in C that creates two tasks by using the `fork` primitive, provided by POSIX [1]. Tasks are synchronized using a producer/consumer process (for the sake of clarity, we suppose that P and V are lock acquisition primitives provided by the platform). In the corresponding annotated AST, all nodes calling a semantic primitive are red while nodes corresponding to semantic values are light gray.



```
1.{
2.   …
3. pid = fork();
4.   …
5. if(pid)
6. {
7.    p(&lock1);
8.      …
9.    v(&lock2);
10. }
11. else
12. {
13.   p(&lock2);
14.     …
15.   v(&lock1);
16. }
17.}
```

Figure 3. Annotated AST

To select a node calling a primitive with the semantic role `CREATE-TASK`, the following expression is used (where `node` ∈ AST).

```
node::semantic::primitive::sem-role==
CREATE-TASK
```

In order to check whether the returned value of the fork is tested, we use the following expression:

```
node::semantic::variable::type == PID-T ∧
node::root::token == IF-BODY
```

The first expression selects a node corresponding to a semantic variables having `PID-T` type. The second expression tests whether this node is in conditional branch which means that the token corresponding to the root of the current node is an `IF-BODY`.

IV. SPECIFICATION OF PROPERTIES

Once the AST is annotated, verification of properties can be performed. Each property is described using a pattern provided by Table I.

TABLE I. A PROPERTY ANALYSIS PATTERN

Identifier	*Property identifier*
Checking process	*Defines the steps of the property verification process. These steps can comprise extraction of various models.*
Property Specification	*Specification of the property*

Several models can be used to check a single property. Steps of the checking process are specified by using the following formalism:

```
step-id: from [quantifier]{input models}
         extract[quantifier]{output model}
         according to{model extractor
                      pattern}
step-id: verify {prop-id} on {model}
```

Property specification is a logical expression that can be stated using first order logic, CTL, LTL, or other specific formalism and that can be verified upon the last extracted model.

V. MODEL EXTRACTOR PATTERNS AND TRANSFORMATION RULES

When stating the checking process within the property analysis pattern, steps refer to model extraction patterns. These patterns define how to build abstract models according to the API semantics. They have the format described in Table II.

TABLE II. MODEL EXTRACTOR PATTERN

Input Models	*Input models from which output models will be extracted*
Output Model	*output model can be referred here, several instances can be extracted.*
Model Transformation Rules	*Model building rule*

A model extraction is based on transformation rules. We introduce how to state the method for specifying pre-conditions and post-conditions constraining initial configurations in input models and resulting configurations in output models respectively.

A. Models transformation rules

Model transformation rules allow deriving a new graph configuration from one or several initial ones.

$$\{G_0,\dots,G_n\} \Rightarrow G_d$$

Transformation rules are based on three elements introduced in Table III.

TABLE III. TRANSFORMATION RULES

Pre-conditions	*Set of configuration rules that are respected by input models*
Nodes building rules	*Algorithm for building sets S, s_0 and s_e*
Transitions building rules	*Algorithm for building the set T*

B. Pre-conditions

In the pre-condition section, the user will define a set of nodes from which the output model will be derived. Pre-conditions are expressed using the following formalism:

```
p-id : { nodes: input-graph | φ(nodes)}
```

`p-id` describes a precondition φ that has to be respected by the set `nodes` ⊆ input-graph ("|" means "such that").
`p-id` is the identifier of the pre-condition. A pre-condition can state, for instance, `"There exists at least one node that is a fork call"` which is expressed as follows:

```
p1: {fork-node ∈ AST |
    (fork-node::identifier==fork)∧
    (fork-node!=0)}
```

C. Nodes building rules

Nodes building rules define `S`, s_0 and s_e of output models according to provided pre-conditions. They are considered as implication rules expressed as follows:

```
{precondition}→{{graph}::set=building-rule}
```

"→" means "implies". Building rules are expressed by using one of the following propositions (n_i, n_j ∈ `{graph}::S` and one of expressions between brackets or even both):

- ***include all[from** n_i **until** n_j**][such that** φ**]**: this rule includes into the specified set all the nodes of the subset specified by the optional expression *[**from** n_0 **until** n_i]* or [**such that** φ];
- ***exclude** all[**from** n_0 until n_i][**such that** φ]*: exclude from the set S nodes of the subset specified by the optional expression *[**from** n_0 **until** n_i]* or [**such the** φ];
- ***build** nodes **according to (f)***: build a node with a new format generated by the function *f* (e.g., from control graph, build a node with task-id).

D. Transition rules

Transition rules define the algorithm for connecting nodes to each other in the output model G_d according to their configuration in the initial model $\{G_0,\dots,G_n\}$. Transition rules are also expressed using:

- Pre-condition: we assume that there exists one or several element $s_i \in S$ that have their projection $s_d \in S_d$, pre-conditions introduce properties that have to be respected by s_i in the initial model G;
- Post-condition: define the type of links that connect S_d nodes in G_d.

Transition rules are expressed as follows:

```
{precondition}→{connection-rule}
```

Where connection-rule have the following format (the fourth optional parameter specifies the transition label):

$$\texttt{connection}(G_d::s_1, G_d::s_2, \texttt{type}, [l])$$

VI. APPLICATION

We illustrate our methodology on the simple example provided in Section III.C.2). Let us suppose that we want to check whether locks are correctly released after their acquisition. We will provide the property pattern analyzer corresponding to this property. Then, we provide model extraction patterns used in the verification process.

A. *CORRECT-LOCKS-USE property analysis pattern*

To check whether acquired locks are always released, we need to first extract control graphs in order to determine which task is using which lock. After that, a lock-use-graph is extracted where nodes represent tasks. These nodes are connected by transitions labeled by lock operations (cf. Figure 4).

The property consists of verifying that each node, which is a source of an arc labeled with `ACQ-LOCK` on a lock, has an entrant arc labeled with `RLS-LOCK` on the same lock.

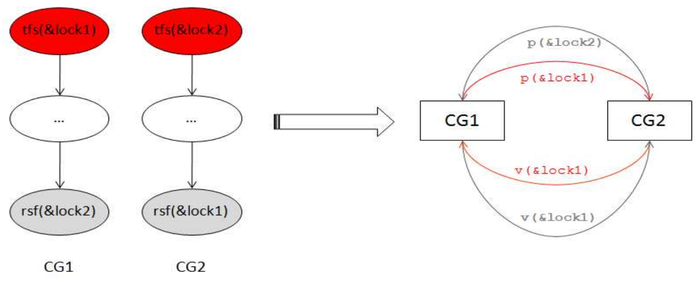


Figure 4. Lock-use-graph extraction

The pattern corresponding to this property is provided in the Table IV.

TABLE IV. CORRECT-LOCKS-USE ANALYSER PROPERTY PATTERN

Identifier	`CORRECT-LOCKS-USE`

Checking Process	`step1 : `**`from`**` {AAST}`
	   `extract all`` {CG:CONTROL-GRAPH}`
	   `according to``{CONTROL-GRAPH-EXTRACTOR-PATTERN}`
	`step2 : `**`from all`**` {CG}`
	   `extract all`` {LG:LOCK-USE-GRAPH}`
	   `according to``{LOCK-USE-GRAPH-EXTRACTOR-PATTERN}`
	`step3 : `**`verify`**` {CORRECT-LOCKS-USE}`
	   `on``{LG}`

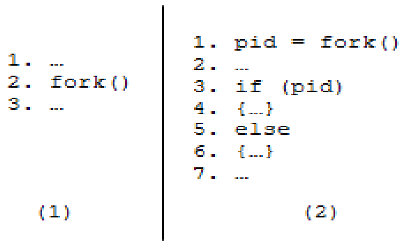
Property Specification	`{∀n ∊ LG::S}`
	`if`
	`{∃t1 ∊ LG::T |`
	    `((t1::org==n) ∧`
	     `(t1::label::sem==ACQ-LOCK))}`
	`then`
	`{∃t2 ∊ LG::T |`
	    `((t2::dest==n) ∧`
	     `(t2::label::sem==RLS-LOCK)∧`
	`(t1::label::param==t2::label::sem))}`
	`else`
	 `Error`

B. *Model extraction patterns*

The property specification pattern refers to two model extraction patterns that are described in the following subsections.

1) *Control graph extraction*

In this example, `fork` primitive is used to create tasks. *`Fork`* is provided by *`POSIX`* and allows the duplication of the current process. *`Fork`* does not take any parameters and returns either 0 (for the child process) or the PID value of the child process (for the parent process). `Fork` can be used differently within a conditional expression or not (cf. Figure 5) and this influences the extraction of the control graph.



```
1. …
2. fork()
3. …

    (1)
```

```
1. pid = fork()
2. …
3. if (pid)
4. {…}
5. else
6. {…}
7. …

        (2)
```

Figure 5. fork writing styles

The first writing style (1) implies that both created tasks have the same behavior and start on the instruction following the call to `fork` (i.e., the instruction in line 3) until the end of the program. In the second case, the behavior of both tasks starts by instructions following `fork` (e.g., instruction in line 2). However, according to the conditional expression that is testing the *`PID`* value, the control graph of the child task continues in else block (i.e., line 6) while the behavior of the parent task is defined by the if block (i.e., line 4). Then, both tasks behavior continues until the end of the program (i.e., line 7).

To specify such semantics, the corresponding pattern described in Table II is expressed as follows:

1. if there exists a conditional expression testing the returned `PID` value; then control graphs `CG1` and `CG2` are created where (1) S_0 refers to the statement following the call to `fork`, (2) S_e points towards the end of the program, (3) S includes S_0, S_e and all nodes following S_0 except those included in IF_`BODY` for `CG1` and those included in `ELSE_EXPR` for `CG2`;
2. if the return value of the `fork` is not tested, both control graphs include all nodes between S_0 and S_e.

A simplified control graph extractor pattern is provided in Table V.

TABLE V. FORK CONTROL GRAPH EXTRACTION PATTERN

Identifier	`FORK-CONTROL-GRAPH-EXTRACTOR`
Input	`AAST`
Output	`CG1, CG2 : CONTROL-GRAPH`
Pre-condition	

```
-- there exists in the AST at least one    --
-- with fork identifier and one node       --
-- with END-OF-PROGRAM node                --
p1:{fork-node, end-node ∊ AAST |
      (fork-node::identifier==fork) ∧
      (end-node::token==END-OF-PROGRAM)}

--  return value of fork is tested         --
p2:{cond, fork-if-body, fork-if-end ∊ AAST |
    (cond::token={VALUE,FUNC-CALL} ∧
     ((cond::semantic::VAR::type=PID-T) ∨
      (cond::identifier==fork))    ∧
     ((fork-if-body::token==IF-BODY-EGIN(cond))
        ∧
      (fork-if-end::token==IF-BODY-END(cond))}
```

-- pre-condition stated to check whether -- -- else exists -- p3:{fork-else-body, fork-end-body ∈ AAST | ((p2::cond)≠ 0 ∧ (fork-else-body::token==ELSE-BODY-BEGIN(cond)) ∧ (fork-else-end::token==ELSE-BODY-END(cond))}}
Node building rules
{p1 ∧ p2}→{{CG1, CG2}::S_0 = NEXT(fork-node) ∧ {CG1, CG2}::S_e = END-OF-PROGRAM} -- both graphs have the same behavior -- {p1 ∧ ¬p2∧ ¬p3}→{{CG1, CG2}::S = include all from S_0 until S_e} -- graphs have different behavior -- {p1 ∧ p2 ∧ p3}→ {CG1::S=**include all from** S_0 **until** S_e ∧ CG1::S=**exclude** all **from** fork-else-body until fork-else-end ∧ CG2::S=**include all from** S_0 **until** S_e ∧ CG2::S=**exclude** all **from** fork-if-body until fork-if-end }
Transition building rules

{ ∃ {n1, n2} ∈ AAST, {gn1, gn2} ∈ CG::S
| gn1=proj(n1), gn2=proj(n2) ∧ next(n1,n2)}
→
{ connect(gn1, gn2, direct-transaction) }

2) *Extraction of the lock use graph*

The lock use graph is extracted according to the pattern provided in the Table VI.

TABLE VI. LOCK USE GRAPH EXTRACTION PATTERN

Identifier	LOCK-USE-GRAPH-EXTRACTOR
Input	CONTROL-GRAPH
Output	LG : LOCK-USE–GRAPH
Pre-condition	
-- there exists a lock acquisition node -- -- in the control graph -- p1:{lock, lock-acq-node ∈ CONTROL-GRAPH | (lock::sem::var::type==LOCK ∧ lock-acq-node::sem::primitive::sem-role == LOCK-ACQ(lock))} -- there exists a lock release node in the -- -- control graph -- P2:{lock, lock-rls-node ∈ CG | (lock::sem::var::type==LOCK ∧ lock-rls-node::sem::primitive::sem-role == LOCK-RLS(lock))}	
Nodes building rules	
{∀cg : CG | (p1 ∨ p2)}→{LG::S = build(identifier(cg))}	
Transitions building rules	
{ ∀ l1, n1 ∈ CG1: CONTROL-GRAPH, ∀ l2, n2 ∈ CG2: CONTROL-GRAPH, ∃ n3,n4 ∈ LG::S | ((n1==lock-acq-node) ∧ (n2==lock-rls-node) ∧ l1::identifier==l2::identifier) ∧ (n3==identifier(CG1) ∧ n4==identifier(CG2)} → { connect(n3,n4,direct,n1) ∧ connect(n4,n3,direct, n2)}	

VII. CONCLUSION AND FUTURE WORKS

This paper deals with the question of how to automatically extract different viewpoint models from source code in order to validate system behavior according to a set of properties. We propose a pattern-based approach that allows specifying the property to check and the transformation rules to apply. For each pattern, a dedicated formalism was introduced. This approach provides more generality than the existing ones. It can be applied for different systems using different languages. Users can plug-in different language parsers and provide the corresponding API semantics.

This approach also allows knowledge capitalization by explicitly defining the verification and transformation processes. It can facilitate verification and validation processes, particularly when these are performed by a third-party organization.

Currently, a prototype was developed allowing checking several design rules such as correct use of locks, atomicity and deadlock. The next step will consist on testing its scalability on great systems.

For future work, we aim to improve our method in order to address temporal constraints. The key point is specifying how to extract a temporal viewpoint model and how to perform the analysis of the satisfaction of temporal constraints.

ACKNOWLEDGMENT

We would like to thank Simon Bliudze for comments on this work.

REFERENCES

[1] POSIX Certification – updated on 3 November 2003 - http://www.opengroup.org/certification/idx/posix.html

[2] OSEK VDX version 3.0.3 – July 2004 - http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=9&Itemid=13.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled, "Model checking", MIT Press, 1999, ISBN 0-262-03270-8.

[4] M. P. Bonacina: "On theorem proving for program checking: historical perspective and recent developments", PPDP 2010, pp. 1-12.

[5] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", Conf. Rec. of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL' 77), ACM Press (New York), Los Angeles, USA, Jan. 1977, pp. 238-252.

[6] K. M. Chandy and J. Misra. "Parallel Program Design: A Foundation". Addison-Wesley, 1988, ISBN 0-201-05866-9.

[7] L. Lamport., "The Temporal Logic of Actions". ACM Trans. on Prog. Lang. and Systems, 1994, pp. 872-923.

[8] Z. Manna and A. Pnueli. "Temporal Verification of Reactive Systems: Safety". Springer-Verlag, New York, 1995, ISBN 0-387-94459-1.

[9] E.A. Emerson, "Temporal and modal logic", Handbook of Theoretical Computer Science, Chapter 16, the MIT Press, 1990, pp. 995-1072.

[10] P. Cousot, R. Cousot, J. Feret, A. Miné, D. Monniaux, L. Mauborgne, X. Rival. "The ASTRÉE Analyzer". ESOP 2005: The European Symposium on Programming, Edinburgh, Scotland, April 2-10, 2005. Lecture Notes in Computer Science 3444, © Springer, Berlin, pp. 21-30.

[11] P.Baudin, A.Pacalet, J.Raguideau, D.Schoen, N.Williams. "CAVEAT : a Tool for Software Validation". In Proceedings of the International Conference on Dependable Systems and Networks (DSN'02), pp. 537-537.

[12] A. Deutsch. *"Static Verification Of Dynamic Properties".* PolySpace Technologies, 27 november 2007, www.polyspace.com

[13] Coverity prevent: Static Source Code Analysis for C and C++, 2008, http://www.coverity.com/library/pdf/coverity_prevent.pdf.

[14] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks", In Proceedings of the Symposium on Operating Systems Principles, October 2003, pp. 237-253.

[15] P. Cousot, R. Cousot, "A gentle introduction to formal verification of computer systems by abstract interpretation". In Logics and Languages for Reliability and Security, J. Esparza, O. Grumberg, & M. Broy (Eds), NATO Science Series III: Computer and Systems Sciences, © IOS Press, 2010, pp. 1-29.

[16] G.S. Avrunin, J.C. Corbett, M.B. Dwyer, C.S. Păsăreanu, S..F. Siegel, "Comparing Finite-State Verification Techniques for Concurrent Software". Technical Report UM-CS-1999-069, Department of Computer Science, University of Massachusetts, 1999.

[17] Sun MicroSystem "Analyzing Program Performance With Sun WorkShop",1999, http://www.atnf.csiro.au/computing/software/sol2docs/manuals/workshop/analyzing/AnalyzingTOC.html

[18] R. C. Seacord. "Secure Coding in C and C++". Addison-Wesley, September 2005. ISBN: 0321335724.

[19] B. Bellay and H. Gall. "A Comparison of Four Reverse Engineering Tools". In Proceedings of the 4th Working Conference on Reverse Engineering (WCRE '97), Washington, DC, USA, 1997. IEEE Computer Society, pp. 2-11.

[20] R. Kollman, P. Selonen, E. Stroulia, T. Syst, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In Proceedings of the 9th Working Conference on Reverse Engineering (WCRE '02), Washington, DC, USA, 2002. IEEE Computer Society, pp. 22-33.

[21] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker Blast: Applications to Software Engineering". *Int. Journal on Software Tools for Technology Transfer*, 9(5-6): pp. 505-525, 2007. Invited to special issue of selected papers from FASE 2004/05.

[22] T. Ball, E. Bounimova, R. Kumar, V. Levin, " SLAM2: Static Driver Verification with Under 4% False Alarms", In Formal Methods in Computer-Aided Design (FMCAD), 2010, pp. 35-42.

Requirements and Solutions for Tool Integration in Software Test Automation

Bernhard Peischl
Softnet Austria
8010 Graz, Austria
bernhard.peischl@soft-net.at

Rudolf Ramler, Thomas Ziebermayr
Software Competence Center Hagenberg
4232 Hagenberg, Austria
{rudolf.ramler, thomas.ziebermayr}@scch.at

Stefan Mohacsi
Siemens IT Solutions and Services
1100 Wien, Austria
stefan.mohacsi@siemens.com

Christoph Preschern
Ranorex GmbH
8053 Graz, Austria
christoph.preschern@ranorex.com

***Abstract*—In this article, we exemplified today's requirements in integrating test automation tools in terms of three integration scenarios combining industrial strength tools in the area of test management, model-based testing and test executionThe article further sketches solutions for the three scenarios by introducing various integration concepts and by discussing their advantages and drawbacks. Based on successful results we propose a framework for test tool integration.**

***Keywords*—software test tools; test automation framework; application integration.**

I. Introduction

The current landscape of solutions for test automation is characterized by a large number of heterogeneous commercial and open source tools. Many of these tools are highly specialized solutions for specific aspects of testing, they focus on different technologies, or they have been designed with certain development and test paradigms in mind. Hence, although there is a large variety of specialized test tools for test case generation, test management, test execution, etc., little support for combining the numerous specialized tools to an integrated solution is offered. In practice, thus, engineers bother about interfacing two or more tools at the technical level rather than being able to integrate and enhance these tools to a custom tool chain that meets the needs of a specific project or organization. Furthermore, besides the provision of technical interfaces between single tools, testing activities require automated support for activities that span across several steps in the testing process and link testing with related activities of software development and project management. Especially with model-based testing gaining momentum, integration requirements have notably increased due to the various ways to represent and evolve test cases in combination with artifacts from requirements engineering, design and development.

From the perspective of test tool vendors and solution providers, the situation is characterized by similar challenges. “80% of the effort Automated Software Quality (ASQ) tool vendors spend today duplicates the work of others, recreating an infrastructure to enable testing and debugging activities. Only 20% of their work produces new function that’s visible and valuable to testers and developers.” [1] Vendors and developers of test tools have recognized the increasing need for integration that allows them to focus on their specific tool competencies, while still being able to offer a comprehensive testing solution to their customers.

The objective of our work, therefore, is the development of integration concepts for test tools that allow connecting tools from different vendors, each specialized on a particular task in test automation, within an extensible test automation framework. In Section 2, we introduce three commercial software test tools from international tool vendors participating in the Softnet Austria Competence Network. Section 3 describes the application scenarios used for exploring the integration requirements. Section 4 summarizes established integration approaches from which we draw in Section 5, where we present and discuss concepts and first solutions. Section 6 summarizes the paper and outlines future work.

II. Test Tool Landscape

To demonstrate and evaluate the proposed integration concepts, we work together with two international companies developing commercial software test tools that, in combination, represent a lateral cut across typical activities in test automation. The following three tools have been involved in the studied scenarios:

- *IDATG* [3] (Integrating Design and Automated Test case Generation) is a tool for generating test data and test cases that has been developed since 1997 by the Siemens Support Center Test in cooperation with universities and the Softnet Austria Competence Network. The IDATG tool supports various approaches for test design and test case generation including equivalence class partitioning, boundary value analysis, cause-effect analysis [2] as well as random and hybrid test case generation [3]. Over the years, the functionality has been continuously expanded and the tool has been successfully applied in numerous commercial and industrial projects within Siemens and by customers such as the European Space Agency ESA. Today, IDATG is a commercial tool offered in combination with the test management solution SiTEMPPO described in the following.
- *SiTEMPPO* [23] is a solution for managing large test case portfolios and related artifacts such as test data, test

results and execution protocols. The tool supports test planning, test case design and specification, the composition of test suites, manual and automated execution of test cases as well as the analysis and reporting of test results [20]. Test management as the coordinating function of software testing interacts with a variety of other development and testing activities such as requirements management, change and defect management and test automation. Hence, the tool has to offer interfaces to a number of related but separate tools for data exchange and synchronization. SiTEMPPO has been developed by an initiative of Siemens Austria. Nowadays, the tool is applied in projects within Siemens all over the world and it is licensed as commercial product for test management on the open market with customers from various industrial domains as well as commercial and public organizations.

- *Ranorex* [24] is a solution for developing and executing automated test cases. The focus of the Ranorex test tool is on the user-friendly capture and replay of robust test scripts building on the accurate recognition and unique identification of user interface elements of applications based on a broad spectrum of different technologies, from C#, VB.NET, WPF, Flex/Flash, to Java and even Qt. The unique strengths of Ranorex's capturing facilities made it a widely recognized test automation tool successfully applied by numerous customers all over the world. The reliable capturing facility allows for an automatic provision of the various elements of the user interface and can thus support the modeling of user interfaces and workflows. Therefore the Ranorex test tool has also been used in a lightweight model-based approach for random test case generation and execution [4]. With the ever increasing variety of user interfaces and the various notification mechanisms in behind, robust replay mechanisms are further an important part in executing and recording the tests being generated.

III. Usage Scenarios and Requirements

In the context of the tools listed above, various usage scenarios have been identified and investigated.

A. *Scenario 1: Test Automation and Execution*

The integration of executable test cases provided, e.g., as test scripts in a test management environment like SiTEMPPO is a vital part of automating the test process. In this scenario, we do not address the generation of test cases but take care of the task of executing the test cases (no matter where the test cases stem from) and recording the results in a test case management tool. This scenario involves several tasks. Typically, for every test case we have to provide test data and the path to the test script for executing the test case. The result of the execution is typically persisted in form of a log file. The test management tool has to access and interpret the log file in order to derive the results of the test case execution.

Although a technical solution for interfacing the tools in this basic scenario can easily be envisioned, when coupling tools of two different vendors, a couple of challenges are involved. For example, how can the message *"testscript foo.bar failed in line 42"* be mapped to a step in the test case specification? What is reported if the test case execution is not terminating or terminates with a timeout? Who should be notified when the test execution failed due to a problem in the setup of the execution environment? Such questions are typical for any integration scenario and illustrate that the various aspects involved have to be addressed at different levels of integration by different integration concepts.

B. *Scenario 2: Model Evolution in Model-based testing*

Model-based testing promises to offer solutions to many of the problems that make software testing a complex task. In theory, given a suitable behavioral model of the SUT, any number of test cases can automatically be generated with respect to planned adequacy criteria and the model serving as a test oracle [7]. To leverage the full potential of test case generation, a complete, detailed and correct model of the SUT – a golden model – has to be provided. Ideally, such a model of the SUT is built on the grounds of requirements or existing specification documents. So the model encodes the intended behavior and can reside at various levels of abstraction [7]. Further models that focus on the workflow and the possible user interactions (e.g., via the GUI elements) may assist in the systematic design of test cases respectively in their automated generation.

Even with considerable upfront investments in terms of resources, time and money, such a golden model can hardly ever be achieved in practice due to several reasons. First, the model needs to capture specific aspects of the SUT at a very detailed level, e.g., GUI elements and workflows. However, in many cases the requirements do not contain the necessary details and, thus, the only options are making adequate assumptions or reverse engineering these missing details by exploring the actual implementation. Second, like programming, modeling is an error-prone task and without frequently executing the model throughout model development, faults in modeling are rather the rule than the exception.

Executable models are known to improve the situation, but are not able to overcome this problem fully. Therefore, tools such as IDATG propose the combination of model-based testing with GUI exploration and capturing techniques employed within capture and replay tools like the Ranorex Studio. This allows for an early detection of faults in the models being developed as test cases, as they can be executed on the GUIs and workflows even in early stages of development when almost no business logic is implemented behind the GUIs. An agile development process, where GUIs - from the very beginning - are crucial elements and are thus directly influencing the modeling process, increase the chance that the software finally will solve the problem of the customer rather than conform to a specification that does not capture the problem in its full shape.

Figure 1 illustrates a scenario for an integrated tool chain. The scenario involves several tools: the Siemens IDATG test case generator, a model editor (e.g., a workflow editor or a UML modeling tool, the IDATG tool comes with its own model editor), the Ranorex GUI spy and the Ranorex replay component. The scenario starts with capturing a specific view of an application (view 1) and continues with recording of a second view of the SUT (view 2). Capturing of the GUIs establishes a rather detailed level of modeling from the very beginning when compared to a purely manual modeling process. The process of capturing introduces conceptual units and recurring building blocks, which support reuse between test cases and even between test cases across different projects.

Afterwards, the result is handed over to a modeling tool where the result of the recording process (view 1 and view 2) is combined and enriched with further details from the requirements document or the knowledge of the SUT. The automated extraction of model components alongside with the composition of components reduces the upfront investments and thus removes a substantial entry barrier into model-based testing also from an economical perspective.

Thereafter the criteria for the test case generation are specified and the model is handed over to the IDATG test case generator for generating the test sequences (which correspond to paths in the model) and corresponding test data. Finally, the Ranorex replay component is employed to execute the generated test cases on the GUI of the SUT.

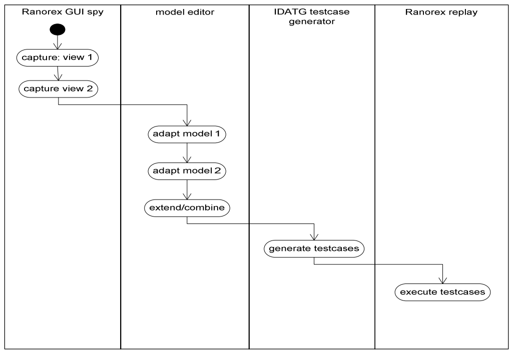


Figure 1: Example workflow with an integrated tool chain.

C. *Scenario 3: Managing Requirements-based Testing*

Testing that the specified requirements have been correctly and completely transferred into executable software is an essential part in the software development lifecycle. In this scenario testing embraces a range of verification and validation activities as well as interfaces linking the results to development and management. In particular, this scenario demonstrates the need to integrate tools across the test and development process to establish a tool chain where the results of one phase build the basis for the next phase. However, the integration is not only characterized by passing on results but includes several update and feedback cycles.

SiTEMPPO supports a requirements-based approach for testing by organizing the test case portfolio according to the structure of the requirements, by tracing test cases to requirements and by reporting test results from the perspective of covered requirements. Figure 2 gives an overview of the involved activities and interactions.

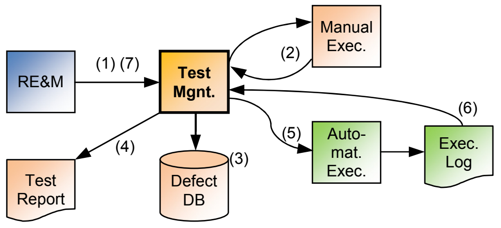


Figure 2: Activities and interactions in requirements-based testing.

(1) Requirement trees are imported into the test management tool as read-only structure. For every imported requirement one or more test cases are derived. The tree structure is used to organize the set of new test cases. Coverage reports show which test cases are linked to requirements and, vice versa, which requirements are covered by test cases.

(2) In a first run, the test cases are executed manually. The test execution results are evaluated and (3) defect reports are issued to a separate defect database when bugs are encountered. (4) Furthermore, the evaluated test execution results are mapped to requirements, indicating that the implementation of a requirement has either been successfully verified or still contains bugs. This first manual run has a strong explorative character and not only focuses on testing the software system but also serves as check whether the requirements have been correctly translated into test cases.

(5) For stable requirements that are subject to ongoing regression testing, test engineers – often located at distributed development sites – automate the manual test cases with tools such as Ranorex Studio or IDATG. The resulting test scripts are linked to the test cases in the test management tool. As described in Scenario 1, SiTEMPPO provides mechanisms for running the test scripts from within the test management environment and (6) for collecting the execution results to evaluate which test cases passed or failed. The results are again mapped to requirements for reporting.

In many projects changing requirements are a constant factor that adds further complexity and dynamics to requirements-based testing. (7) Changes in the requirements have to be propagated to the derived test cases and, furthermore, to the associated test scripts. Keeping requirements, test cases and test scripts synchronized requires coordination and collaboration between the different roles such as requirements analyst, test manager and test engineer. However, without appropriate mechanisms incorporated in test and development tools, coordination and collaboration becomes an ever increasing challenge for distributed teams.

While most of today's tools lack support for coordination and collaboration, SiTEMPPO already includes basic mechanisms like versioning of test cases and linking execution results to the corresponding version of a test set. Nevertheless, as users demand short feedback cycles and constantly up-to-date information on the status and progress of testing across all involved roles and activities, future solutions need to close the currently existing gap between the different tools at the process level.

Layer	Technology	Communication	Coupling	Interaction dynamics	Transformation	Usage context
Business process	**Workflow Engine**	both	transparent	business rules	low	multi role
Application	**Service Bus**	both	transparent	business rules	yes	multi application
	Messaging	asynchron	transparent	registration	low	multi application
	Service	synchron	loose	registration/broker	low	single application
	Business components/RPC	both	strong	static	low	single application
	Shared library/API	synchron	strong	static/plug-in	low	single application
Data	**Database**	both	strong	static	low	single/multi app.
	File	both	strong	static	possible (XSLT)	single application

TABLE I. OVERVIEW OF INTEGRATION CONCPETS

IV. INTEGRATION CONCEPTS

The area of Enterprise Application Integration (EAI) has a long history in developing integration concepts for interaction between existing functionality. Approaches for integration can be categorized by the architectural level where the integration is established [11] or by the communication paradigm underlying the integration [12]. We adopted the categories proposed in literature and summarized the existing integration concepts in Table 1.

In Table 1, the column *Layer* indicates the architectural layer at which the integration is taking place. *Technology* names the commonly applied technologies used for integration. *Communication* shows whether the possible communication options are synchronous, asynchronous or both. The column *Coupling* indicates the strength of the connection between the integrated applications. *Interaction dynamics* states whether the integration is static or dynamic, i.e., has to be set up before the start of the application or can be established and changed at runtime. Data *Transformation* is an important aspect for data exchange between applications and is therefore supported by some of the listed integration concepts. *Usage context* indicates from the user perspective whether integration is possible with one or more other applications.

In the following, the integration concepts as presented in Table 1 are briefly described.

- **File** and **Database:** Integration at the lowest architectural level, the data level, allows the exchange of data between otherwise heterogeneous applications. Data level integration can be implemented in various ways, e.g., by file exchange, by sharing a database, or by copying data from one database to another [15]. This approach may include data transformation if data structures are not compatible. File data might be structured as XML data which provide a stable basis for data exchange and transformation, e.g., using XSLT. While the communication at data level is often easy to implement and has minimal impact on the existing applications, the main drawback of this level is that the applications' existing functionality is not integrated and therefore not reused. Redundant implementations of the same functionality may lead to an increased development and maintenance effort and, furthermore, increases the risk of incompatibility between applications.
- **Shared Library** and **Application Programming Interface (API):** Good software design encourages the reuse of existing implementations, e.g., provided as components in a shared library or in form of plugins. Interfaces encapsulate the functionality and implementation. Via interfaces the functionality of other applications can be accessed. Integration at application interface level (see [11]) can be implemented at different abstraction levels like integration of data access functionality or integration of functionality that contains business logic. Integration at this level leads to strong coupling between applications. Transformation is not supported by default and it supports integration with a single other application. However, if an application already provides an API, implementing integration at this level is easily achievable even without additional infrastructure.
- **Business components, Remote Procedure Calls (RPC):** At higher abstraction levels an application may consist of business components that provide rather coarse-grained business functionality [13]. This functionality can be integrated in other applications in various ways, either by packing them to the application where they should be integrated or by remote procedure calls. Using business components remotely requires that the remote application is running. Business components provide the highest functional abstraction level of an application and, therefore, reuse at the highest functional level. Coupling at this level is strong and transformation support not natively built in. Yet the functional reuse level is high.
- **Service:** Software services provide means for loose coupling of applications as they encapsulate functionality and the site where this functionality is running [14]. The concept of Web services provides standardized protocols for communication to integrate applications across platform borders. Overall, integration at service level means integration at a coarse-grained business function level for reusing application functionality at business level. The advantage of this level is the loose coupling and mostly standardized communication protocols, but without additional infrastructure, communication is still synchronous without transformation support and it is used for integration with a single application.
- **Messaging:** In some cases asynchronous communication is required due to performance reasons or the need

for weak coupling. These requirements can be addressed by a message queue which decouples communication partners in a timely manner and provides guaranteed message delivery. Message queues are also used for sending messages to multiple applications (broadcast), with or without feedback about delivery; see integration styles in [6]. Advantages of this integration level are asynchronous communication, low coupling and integration possible with multiple applications.

- **Service Bus:** A common integration concept is the service bus which provides functional support for the integration and communication between applications. The idea is to connect all applications with a bus where applications put messages on the bus and others listen and take the messages relevant to them. A service bus also supports plugging in additional components like transformation or filter components that allow modifying or removing messages. Furthermore, some service bus implementations support defining message flows between applications and components including splits and joins [6]. This integration level supports all features presented in Table 1 except the possibility of integration along a workflow involving responsibilities and roles.
- **Workflow Engine:** From a user perspective, the usage of applications follows organizational workflows which define task order and responsibilities. In order to accomplish the work, a workflow might contain multiple tasks that utilize different applications. From a technical perspective, a sequence flow between tasks utilizing different applications indicates integration of those applications (see also [16]). Workflow engines are able to implement communication at workflow level and coordinating the use of applications integrated at a technical level. This is the highest and most abstract integration level with support for transparent coupling, dynamic interaction based on business rules and integration of the work processed by multiple roles.

V. Solutions and Discussion

This section describes and discusses how the integration requirements elaborated from the usage scenarios in Section III can be supported by the technologies presented in Section IV. The integration concepts have been explored either via a (prototypical) implementation or a design study elaborated together with developers and architects of the test tools.

A. *Scenario 1: Test Automation and Execution*

In coupling the SiTEMPPO test management solution with the Ranorex test automation tool, we follow the paradigm of a strong coupling with the need for both, asynchronous and synchronous communication. Due to the specialized interface, the interaction dynamics remains static without the need for transformations. Thus, the integration is established via files, i.e., at the level of the data layer (Table I). In detail, the prototypical integration of the SiTEMPPO and Ranorex tools has been implemented as follows:

Ranorex Studio allows creating executable test suites. When the execution of a set of automated tests is triggered in SiTEMPPO, Ranorex Test Runner is called for each test case, passing the name of the corresponding test script as command line parameter. The execution generates a log file in a predefined directory, which is processed by an import adapter implemented as part of SiTEMPPO. The adapter extracts the information relevant for deciding on the test result (passed, failed or blocked).

In order to access the Ranorex tool from within SiTEMPPO, several global settings have to be made in the configuration of the test management environment, e.g., the path to the executable test scripts, the execution log, and the necessary runtime libraries. Hence, the interface implementation part of SiTEMPPO requires exception handling strategies to deal with erroneous configurations and timeouts. Additional setup, rollback and restart mechanisms need to be included in the automated test scripts. Furthermore, predefined execution orders due to implicit dependencies between test scripts cannot be handled by SiTEMPPO.

The benefit of the low-level, static coupling between the two tools is the straightforward implementation of the interface and the ability to consider tool-specific extensions. This benefit turns into a drawback as soon as interfaces for several different test automation tools should be provided. Developing and maintaining a large set of interfaces is cumbersome as the external interfaces may change without notice whenever a new version of an integrated tool is released.

Our experience with implementations for this scenario showed that the initial use case also stretches into the organization dimension. While in an ideal setting the test management supervises the top-down development of automated test scripts from previously defined and specified test cases, in practice, many valuable test scripts also emerge bottom-up and need to be incorporated into the managed test structure. Gathering existing test cases and keeping them synchronized results in a considerable effort for test managers, especially in a distributed project setting. Hence, the need for tool support for discovering and "importing" existing test cases soon appeared as additional requirement. As a consequence we propose an approach emphasizing the inversion of control – developers of automated test scripts should register the new or changed test cases with test management. The responsibility to maintain and update the test cases remains with the test script developers. Integration concepts that support this approach are presented and discussed as part of Scenario 3, Section C.

B. *Scenario 2: Model Evolution in Model-based Testing*

A key requirement for our Scenario 2 is the interaction dynamics. Any solution has to guarantee acceptable response times and ease of use in switching from one tool to the other. Thus, we favor synchronous communication mechanisms and no or rather low need for transformations. There are no multiple roles involved and the interaction happens always between two tools. Thus we propose shared libraries, business components, and plug-ins to implement Scenario 2.

Plug-ins are a common mechanism for adding third-party tools to a tool suite. A plug-in explicitly provides information about its dependencies on other plug-ins. Furthermore, a plug-in can change menus and menu entries as well as popup menus and toolbars. Additionally, it is possible for plug-ins to notice the execution of menu actions of other plug-ins [8]. For these reasons, a plug-in mechanism is very well suited to implement the desired coupling on the application level.

According to [9], a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Besides the specification of provided interfaces, the definition of a component also requires components to specify their needs. In other words, a component requires a specification of what the deployment environment will need to provide such that the component will operate. In principle, this is a generalization of plug-ins and might thus be appropriate for implementing the coupling as well.

Software engineers in general create many scenarios (and corresponding model artifacts) and often recall prior work as they develop models for novel use cases. The process of refinding patterns is a popular approach in this respect and will be supported by the concept of a shared library. However, the adequate abstraction level of the models (or building blocks) stored in the library is a challenging research issue. Basically, we pursue two main directions in supporting reusability: tagging and structural similarity [5].

C. Scenario 3: Managing Requirements-based Testing

The third scenario is characterized by the need for integration at the process level to support coordination and collaboration across different roles, phases and distributed development sites. Conventional approaches rely on a central coordination instance, usually represented by test management. In that constellation the test management tool is used as central hub, gathering and consolidating information from the various other test tools. Technically, the interfaces between the involved tools remain on the lowest level; mainly data exchange via import/export facilities is supported.

The specialization of the different tools is generally quoted as reason why sharing functionality between tools is insufficiently attractive. However, the numerous redundant features provided by the different tools reveal that the opposite is true. For example, almost all tools implement their own reporting. The slight but obvious variances in the reporting of the different tools are a common nuisance for users, especially when they try to analyze the status of testing over all activities from data spread across different tools. As a result, existing reporting facilities are once more implemented as part of test management tools in an attempt to create a homogeneous, aggregated view on the test process.

With the test management tool as central hub and all other tools arranged as satellites, the management tool becomes the bottleneck in the test tool infrastructure. It has to provide interfaces to all tools included in testing and, thus, the provided interfaces are the main limitation in the choice of applicable tools. Projects suffer from this inflexibility when the optimal test tool cannot be applied due to test management not offering the corresponding interface or – in case generic adapters exist – when test management lacks the resources to setup and maintain the necessary interface configurations. Moreover, the strong coupling of the data level integration turns intro rigid dependencies. Even minor changes in the data format may render the interface incompatible. Hence, in practice, many projects are tied to outdated versions of tools because of update incompatibilities. Tool providers, however, often do not even know about the potential conflicts since they are not aware of the dependencies to the interface implemented as part of the test management tool.

As indicated in Scenario 1, Section A, we propose to emphasize the Inversion of Control principle for tool integration at the process level. Test management has to be released from the burden of gathering and extracting data from the various other test tools. In contrast, the satellite tools have to take over the responsibility of providing the necessary data and maintaining compatibility. Now, however, instead of test tools interfacing directly with various different test management tools resulting in a complex point to point integration, the tool communication should be extracted into a separate integration facility serving as backbone of the tool infrastructure. Service-oriented concepts have been proposed and were successfully evaluated for software engineering environments [17]. Drawing from positive experience with integrating software engineering tools, we adopted the service bus approach (Figure 2) specifically for test tools.

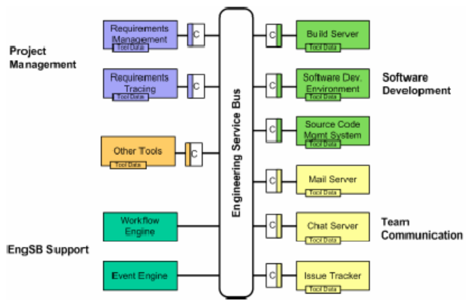


Figure 3: Engineering Service Bus (EngSB) for integrating software engineering tools [17].

The illustrated approach enables communication between the different tools beyond the level of data exchange. Status messages can be exchanged to notify other tools about completed activities and pending updates. For example, test execution tools can send a message indicating the successful completion of a test run. The message can include relevant result information and a link to the execution log. Thus, instead of storing static configuration details such as the location of execution logs in the test management tool's settings, all concerned tools register for the corresponding message and receive the information at runtime. Furthermore, the link may not point to a static location from where the log is retrieved as file, but to a service interface that allows querying and analyzing relevant aspects of the execution. Providing the query logic as a service of the execution tool avoids redundant implementation of analysis functions.

A prerequisite for the service-based integration of tools is the agreement about offered services, data structures and exchange formats. In software and systems engineering and in particular in testing, several relevant standards are in place, for example the UML Testing Profile [21, 22], the IEEE Std. 829-2008 for Software and System Test Documentation, or the Requirements Interchange Format [18].

Furthermore, automated transformation of messages, models and data formats implemented in form of services are also connected to the service bus.

Communication and teamwork requirements can be addressed by adding shared services for reporting, monitoring, status notification and even workflow-based collaboration. An example for a tool providing shared services is a test cockpit [19] providing insight on the status and progress of testing across all involved roles and activities.

VI. CONCLUSION AND FUTURE WORK

In this article, we exemplified today's requirements in integrating test automation tools in terms of three integration scenarios combining industrial strength tools in the area of test management, model-based testing and test execution: The test and requirements management tool SiTEMPPO, the Siemens IDATG tool for model-based testing, and the Ranorex automation tool suite. The integration scenarios represent typical situations frequently encountered in real-world projects by the authors: (1) Combining test automation and test execution, (2) model development and evolution in model-based testing, and (3) the management of requirements-based testing and regression testing. For each of these scenarios, solution concepts have been developed and explored together with developers and architects of the presented tools, based on existing integration technologies (file-level data exchange, plug-in concept, messaging and service bus). It could be shown that the elicited integration requirements of each scenario can be addressed by applying existing concepts, which are attributed the potential for building a framework able to combine a set of heterogeneous tools by different vendors. Although the higher-level integration concepts show a larger potential w.r.t. integrating heterogeneous tools, we also found that no single integration concept is able to cover all requirements from the explored scenarios.

Our next step will be to consolidate the existing implementations and concepts towards a service-oriented integration platform easily extendable by future test and development tools.

ACKNOWLEDGMENT

The research herein is partially conducted within the competence network Softnet Austria II (www.soft-net.at, COMET K-Projekt) and funded by the Austrian Federal Ministry of Economy, Family and Youth (bmwfj), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

REFERENCES

[1] The Hyades Project Automated Software Quality for Eclipse: http://www.eclipse.org/tptp/home/archives/hyades/project_info/HyadesFormation.12.pdf, last visited on 27th June 2011.

[2] A. Beer and S. Mohacsi, “Efficient Test Data Generation for Variables with Complex Dependencies”, Int. Conf. on Software Testing, Verification, and Validation, 2008, pp. 3-11.

[3] S. Mohacsi and J. Wallner, “A Hybrid Approach for Model-Based Random Testing”, in Advances in System Testing and Validation Lifecycle (VALID), 2010, pp.10-15, 22-27 Aug. 2010

[4] B. Hofer, B. Peischl, and F. Wotawa, “GUI Savvy End-to-End Testing with Smart Monkeys”, Fourth International Workshop on the Automation of Software Test, Vancouver, Canada, May 16-24, 2009.

[5] W.N. Robinson and H.G Woo,“Finding reusable UML sequence diagrams automatically”, IEEE Software, vol. 21, no. 5, pp. 60- 67, Sept.-Oct. 2004.

[6] G. Hohpe and B. Woolf, “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions”, Addison-Wesley Professional, 2003.

[7] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches”, Software Testing, Verification and Reliability, 2011. Published online, paper version in press.

[8] S. Burmester et al., “Tool integration at the meta-model level: the Fujaba approach”, Int. J. Softw. Tools Technol. Transf. 6, 3 (August 2004), pp. 203-218.

[9] C. Szyperski, “Component Software: Beyond Object-Oriented Programming”, 2nd ed. Addison-Wesley Professional, Boston ISBN 0-201-74572-0.

[10] Eclipse TPTP, Eclipse Test & Performance Tools Platform Project,http://www.eclipse.org/tptp/, last visited 27th July 2011.

[11] D.S. Linthicum, “Enterprise Application Integration”, Addison-Wesley Professional, 1999, ISBN.: 978-0-201-61583-8.

[12] Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, 2003.

[13] P. Herzum and O. Sims, “Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise”, John Wiley & Sons, New York, NY, USA 2000, ISBN:0471327603.

[14] T. Erl, “Service-Oriented Architecture: Concepts, Technology, and Design”, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[15] M. Vujasinovic and Z. Marjanovic, “Data Level Enterprise Applications Integration”, in Business Process Management Workshops, pp. 390-395, Volume 3812, Lecture Notes in Computer Science 2006, Springer.

[16] J.A. Espinosa and A. Sanz Pulido, “IB (Integrated Business): A Workflow-Based Integration Approach”, Hawaii International Conference on System Sciences (HICCS), 2002.

[17] S. Biffl and A. Schatten, “A Platform for Service-Oriented Integration of Software Engineering Environments”, 8th International Conference on Software Methodologies, Tools and Techniques (SOMET 09), 2009.

[18] M. Jastram and A. Graf, “Requirements, Traceability and DSLs in Eclipse with the Requirements Interchange Format (RIF/ReqIF)”, Dagstuhl-Workshop MBEES 2011: Modellbasierte Entwicklung eingebetteter Systeme, 2011.

[19] S. Larndorfer, R. Ramler, and C. Buchwiser, “Experiences and results from establishing a software cockpit at BMD Systemhaus”, 35th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA 2009), pp. 188-194, IEEE, 2009.

[20] R. Ramler, G. Czech, and D. Schlosser, “Unit Testing beyond a Bar in Green and Red”. 4th int. Conf. on Extreme Programming and Agile Processes in Software Engineering, XP 2003.

[21] OMG, “UML testing profile Version 1.0”, OMG, 2005. formal/05-07-07; http://utp.omg.org/.

[22] P. Baker, Z. Ru Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, “Model-Driven Testing: Using the UML Testing Profile”, Springer, 2007.

[23] SiTEMPPO: www.siemens.at/sitemppo, visited 27th July 2011.

[24] Ranorex Automation Studio: www.ranorex.at, visited 27th July 2011.

.

RobusTest: Towards a Framework for Automated Testing of Robustness in Software

Ali Shahrokni, Robert Feldt
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
ali.shahrokni, robert.feldt@chalmers.se

***Abstract*—Growing complexity of software systems and increasing demand for higher quality systems has resulted in more focus on software robustness in academia and research. By increasing the robustness of a software many failures which decrease the quality of the system can be avoided or masked. When it comes to specification, testing and assessing software robustness in an efficient manner the methods and techniques are not mature yet.**

This paper presents the idea of a framework RobusTest for testing robustness properties of a system with focus on timing issues. The test cases provided by the framework are formulated as properties with strong links to robustness requirements. These requirements are categorized into patterns as specified in the ROAST framework for specifying and eliciting robustness requirements. The properties are then used for automatically generating robustness test cases and assessing the results.

***Index Terms*—Robustness, real time systems, testing, timing**

I. Introduction

Robustness is an essential software quality attribute that is defined as [1]:

> The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

Timing properties of software have a major role in determining the degree of robustness of the system. In our previous work [2] we focused on elicitation and specification of robustness requirements. In that study we have categorized requirements for developing a robust system in form of patterns. The motivation for using patterns is to capture the commonalities in structure and purpose of the requirements. The patterns can in general be divided into requirement patterns with the main focus to detect and solve robustness issues intrinsically and patterns that provide extrinsic architectural and design means to prevent robustness issues to surface. The patterns that provide intrinsic robustness are mainly to assure input stability or execution stability of the software. Input and events can be erroneous in two main manners, which can cause instability in systems, incorrect value or incorrect timing. The majority of academic research on robustness has so far been focused on stability of the system given erroneous input [3], [4], [5], [6], [7]. This paper discusses the stability in the presence of input and events with invalid timing. In ROAST, there are seven different patterns that focus on this problem area.

Robustness testing tools for generating random test data such as Ballista [3] and JCrasher [4] are the most well known methods of testing robustness of software systems. Using these frameworks help the user to assess how the system behaves in presence of input with invalid value. These frameworks are automated and the user has very little power to specify what data to test and what the expected result is. Instead, they use simple oracle frameworks such as CRASH [8], which is introduced later in this paper to determine whether the randomly generated input data results in failure in the system. Moreover, there is no or very little focus on the timing aspects of the input data in these framework.

Another framework which works on specifying and testing timing properties of a system is called Timed Input Output Automata (TIOA). With TIOA the user can model the interfaces of the system and specify the expected time intervals for the communications and between the different states of the system [9]. This model can then be used to automatically generate random test cases. To use TIOA the user often needs to create a sequential and large model of the system. Furthermore, when it comes to testing, TIOA mostly focuses on testing for timeout and has no or very little focus on other causes or patterns that can create robustness issues [9].

This paper presents the structure of RobusTest, which is a framework included in the ROAST framework for testing robustness requirements with timing focus. By writing testable properties, RobusTest automatically generates robustness test cases. If specified in sufficient detail these properties can be used as an oracle for the test cases. If there is no specification of the expected behavior, RobusTest oracle will assure that the test case will not put the system in a state with catastrophic consequences using the CRASH benchmarking framework. RobusTest not only provides to a large extent automatic testing but also a strong traceability between the generated test cases and the requirements through properties.

Section II discusses some of the concepts used to build the RobusTest framework. In Section III, we present the RobusTest patterns dealing with timing issues, test case generation, executor and oracle included in RobusTest. Finally, Section V discusses the current and future work planned for RobusTest.

II. Background

In the first part of this section, the robustness benchmarking CRASH is introduced. The second part discusses the concept of property based testing and some of the techniques and tools available for this topic. Both these concepts are used in the framework RobusTest.

A. CRASH

The CRASH framework for benchmarking the robustness of operating systems (OS) was introduced by Koopman and is described in [8]. This framework acted as a simple oracle where the availability of functionality and the functions in the system rather than the correct functionality after the occurrence of a robustness issue can be measured. CRASH is used as the default oracle built in to the RobusTest framework. The CRASH framework is explained below, which will be implemented in our solution as an underlying layer to the framework. However, using RobusTest framework the expected functionality can be specified and benchmark on top of the extent of functionality.

C	Catastrophic (OS crashed multiple tasks affected)
R	Restart (task/process hangs, requiring restart)
A	Abort (task/process aborts, e.g., segmentation
S	Silent (no error code returned when one should be)
H	Hindering (incorrect error code returned)

Catastrophic class occurs when a fault in a part of the system under test (SUT) results in failure in other parts or even crash or hanging of the whole SUT. It usually requires hardware or software restart of the SUT. The Restart class occurs when one task hangs and can be resolved by killing or restarting that task. The Abort class occurs when a single task is abnormally terminated. The Silent class occurs when invalid parameters are submitted, but neither an error return code nor other task failure is generated. The Hindering type of failures is caused when the diagnosis is incorrect and could cause incorrect recovery.

B. Property based testing

A property is a statement which specifies how a system should or should not behave in a specific situation [10] in contrast to a test case that is set of executions done in a certain order. Using property based testing (PBT), high level properties of the system that should hold are stated and they are used to generate test cases in order to verify and validate a certain aspect or property of the system. In PBT a property is specified in a low level specification language. A PBT specification language should provide temporal and logical operators and location specifiers to the tester [10]. This specification written is then used to automatically generate test cases for that property. The expected behavior of the system is also specified in the property specification that can be used by the oracle to automatically analyze the results from the test execution.

Property-based testing intends to establish formal validation results through testing. "To validate that a program satisfies a property, the property must hold whenever the program is executed. Property-based testing assumes that the specified property captures everything of interest in the program, because the testing only validates that property" [10]. Notice that property based testing is in the same way as robustness testing a complementary to other types of verification and validation activities.

Fink et al. have used property based testing to identify security flaws and vulnerabilities in critical Unix programs such as *sendmail* [11], [12]. In the recent years, this type of testing has received increasing attention from the industry and research community. One example is the ProTest project[1] financed by the European Commission to improve methods and tools for property based testing. A well known tool for property testing is QuickCheck, which was initially developed for functional programming languages such as Haskell and Erlang but has now been developed for Java and other languages [13]. An example property written in QuickCheck for testing the functionality to reverse a list of integers can look like this [14]:

$$\begin{aligned}&prop\_reverse()->\\&\quad ?FORALL(L, list(int()),\\&\qquad\qquad reverse(reverse(L)) == L).\end{aligned}$$

Another relevant study for this paper that uses property based testing for verification of the timing properties of an instant messaging server is presented in [14]. Using the Erlang language Hughes et al. generate test cases with a timing focus on an instant messaging server and compare the results of a property based approach with state-machine approach. However, this study has no focus on robustness testing and argues how property based testing can perform well for testing the timing aspects of a system. The timing parts here are mostly focused on timeout and not other timing aspects.

III. Design of RobusTest

This section will discuss the design of the framework RobusTest for testing robustness with focus on timing properties. As discussed earlier there are seven patterns in ROAST with timing focus that are also used in RobusTest since RobusTest is a part of the ROAST framework. These are presented in this section and the first two are discussed in more detail. The other patterns are presented and discussed in less detail.

Figure 1 shows the overall structure of the RobusTest framework. The patterns discussed in this paper and [2] will give a structure to the requirements on the SUT that will be used by the testers to specify properties. These properties are then used by the test generator (TG) to generate test cases automatically. The number of test cases generated can be set manually by the tester. These test cases are then used by the test executor (TE) on the SUT. The results from execution of the tests are sent to the test oracle (TO) for assessment. Assessment is done based on the expected behavior in the properties, the results from running the test cases and the CRASH benchmarking framework.

[1]http://www.protest-project.eu/

In order to analyze the robustness of the SUT expected behavior and response should be specified. However, in some cases it might be enough to check whether running the test case has any critical effect on the SUT in form of crash or restart. The functionality to detect these kinds of failure is therefore built into the framework. Using the framework without specifying the expected behavior can in this way detect the most critical failures. This part of the framework is implemented using the CRASH benchmarking framework. CRASH is built in to the TO as the default oracle. However, CRASH can be supplanted by the expected behavior if provided in the property and it can even be disabled to generate some types of faults if that type of fault such as a system restart is an expected behavior of SUT.

In addition to the automated part of the oracle, it is possible to specify a concrete expected result for the test cases. The mechanism for generating test case and analyzing the result for the two first patterns is given below.

Another important aspect of RobusTest is the alignment of requirements and test cases. This is one of the main focuses of ROAST. Traceability between the requirements specified for the SUT based on the RobusTest patterns and the test cases generated by RobusTest is ensured through properties.

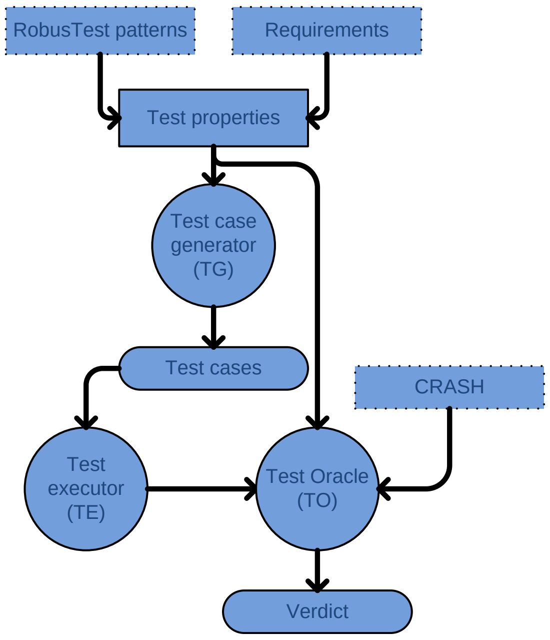


Fig. 1. RobusTest framework structure: The circles are parts of RobusTest and the rectangles are resources provided to or generated by RobusTest. The round rectangles are resources generated by RobusTest, the dotted rectangles are already available and the rectangle with a solid border should be specified by the tester.

A. *Specified response to timeout and latency*

This pattern specifies the expected behavior of the SUT in case an input or event is not received by an expected timeout deadline. To specify a property for this pattern the following factors can be specified: IUT, t_0, t_T, S_I, E, S_E The description of all the parameters in this section is given in Table I.

Given these parameters the timeout property can be specified. This specification can then be used to generate test cases that are run both before and after the timeout deadline to compare the behavior of the SUT in case of timeout with the case when the input is received on time.

S_I is used to specify what the test case needs to perform in order to have the SUT in the appropriate state for starting the test case. S_E is the expected SUT state after the test cases are run.

To generate test cases for this pattern not only the input arriving after the timeout deadline needs to be tested but in some cases test cases with input arriving before the deadline and very close to the deadline need to be generated to have a better understanding of the SUT's timing behavior. Test cases need to set the state of the SUT to S_I and simulate the input or event E. The SUT is then expected to be in state S_E after these actions.

To generate and analyze test cases for timeout the following algorithms are used. If the expected result S_E is not specified those steps with S_E will be neglected. The following is the description of three algorithms for three possible cases occurring when testing this pattern[2]

Input with timing before the timeout deadline:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t_T - \delta < t < t_T$ starting on t_0. (TG)
3. Send a valid input according to the description in E to IUT. (TG)
4. Wait for output from IUT (TE)
   4.1. CRASH $\rightarrow$ Fail (TO)
   4.2. Invalid output according to $S_E \rightarrow$ Fail (TO)
   4.3. Valid output according to $S_E \rightarrow$ Pass (TO)

Input or event not received on deadline:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t > t_T$ starting on t_0. (TG)
3. On time t_T: (TE)
   3.1. CRASH $\rightarrow$ Fail (TO)
   3.2. If the behavior of the SUT is according to $S_E \rightarrow$ Pass (TO)

Input is sent after the deadline t_T:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t > t_T$ starting on t_0. (TG)
3. Send a valid input according to the description in E to IUT. (TG)
4. Wait for output from IUT. (TE)
   4.1. CRASH $\rightarrow$ Fail (TO)
   4.2. Invalid output according to $S_E \rightarrow$ Fail (TO)
   4.3. Valid output according to $S_E \rightarrow$ Pass (TO)

The first algorithm analyzes the behavior of the SUT in cases where the input or event happens very close to the deadline. The purpose for this step is to ensure the correct behavior

[2]The letters in front of each step indicate what part of the framework is responsible for executing that step.

TABLE I
DESCRIPTION OF PARAMETERS THAT NEED TO BE SPECIFIED FOR TEST CASE GENERATION AND ANALYSIS

Parameter	Description
IUT	The set of interface(s) under test.
t_0	The reference time from when the timer should start counting. This is usually connected to an event in the form of an input received or an event in the execution environment.
t_T	The amount of time after the reference time until timeout occurs.
S_I	The initial state of the SUT at the reference time.
E	The expected input.
S_E	The expected behavior and response of the SUT. This might be as simple as the SUT not having any of the CRASH states. It can even be a specified expected behavior such as receiving a specific error message in case of timeout.
f	The maximum acceptable output or input frequency.
E_F	A follow up set of inputs that are dependent on E that will generate faults in the SUT if received before E.

of the SUT in general when the timeout is not expected to happen. The second algorithm assesses the behavior in case a deadline happens when the SUT is supposed to detect the deadline and take appropriate measures to ensure the rest of the functionality is not affected by the omitted input or event. The last algorithm is for generating an input or event after the deadline has been reached. Since the SUT does not have control on how the external parts behave and can not normally avoid them sending inputs or events, this part makes sure that receiving messages after deadline does not affect the correct functionality of the SUT.

A property for this pattern is specified in the following way[3]:

$$\forall t_T - \delta < t < t_T + \delta : setState(S_I, \{t_0\}), timeout \\ (Event(E, \{IUT\}), t) \rightarrow Expected(\{S_{Eb}\}, \{S_{Et}\}, \{S_{Ea}\})$$

where $setState(S_I, \{t_0\})$ specifies that the state of the SUT should be set to S_I at time t_0. $timeout(Event(E, \{IUT\}), t)$ specifies that the timeout happens at time t. $Expected(\{S_{Eb}\}, \{S_{Et}\}, \{S_{Ea}\})$ specifies that the expected behavior upon receiving the event E before timeout is S_{Eb}, after timeout S_{Et}, and the expected behavior in case timeout occurs is specified by S_{Et}.

B. Specified response to input with unexpected timing

This pattern represents cases where an input or event is received while it was not expected. The reason for too early input can be either a missing event that causes irregularities in the reception sequence or input that causes out of order events or inputs or the SUT not being ready to handle the event or input. To specify a property for this pattern the following factors are to be specified: IUT, t_0, S_I, E, S_E.

The difference between this pattern and the timeout pattern is that in this case there can be a need for more thorough understanding of the SUT as a whole. Modeling the SUT or at least parts of it is in some cases inevitable where we want automated generation of test cases for this pattern while in the case of timeout it can be enough to specify a property on a specific interface. The reason is that the initial state is more complex and simulating missing inputs is a more troublesome work than generating a timeout.

[3]Parameters inside {} are optional in RobusTest.

To generate test cases the SUT is configured to S_I after which the event or input E occurs. In the same manner as the previous pattern this needs to be tested on both sides of the deadline. S_E specifies in what state the SUT needs to be after this test in each case.

To generate and analyze test cases for inputs and events occurring with unexpected timing and specially too early inputs the following algorithms are built into RobusTest. Similar to the previous pattern, if the expected result S_E is not specified steps including S_E will be ignored by RobusTest and the default oracle (CRASH) is used.

Input with timing after t_0:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t_0 < t < t_0 + \delta$. (TG)
3. Send a valid input according to the description in E to IUT. (TG)
4. Wait for output from IUT. (TE)
   4.1. CRASH $\rightarrow$ Fail (TO)
   4.2. Invalid output according to $S_E \rightarrow$ Fail (TO)
   4.3. Valid output according to $S_E \rightarrow$ Pass (TO)

Input or event has been received before the starting time t_0:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t < t_0$. (TG)
3. Send a valid input according to the description in E to IUT. (TG)
4. Wait for output from IUT for a certain amount of time for output: (TE)
   4.1. CRASH $\rightarrow$ Fail (TO)
   4.2. Invalid output according to $S_E \rightarrow$ Fail (TO)
   4.3. Valid error message output according to $S_E \rightarrow$ Pass (TO)
   4.4. No output received $\rightarrow$ Pass (TO)
5. When IUT is ready to receive messages if the state of the SUT is incorrect $\rightarrow$ Fail

The first algorithm checks the behavior of the system for inputs received a short while after the SUT is ready to process inputs, while the second algorithm checks the behavior when the SUT is not ready yet to receive any input. This way it is possible to check the behavior of the SUT in cases which should not happen and timings that are very close to the acceptable limit.

A property for this pattern is specified in the following way:

$$\forall t_0 - \delta < t < t_0 + \delta : setState(S_I, t), earlyEvent \\ (Event(E, \{IUT\}), t) \rightarrow Expected(\{S_{Eb}\}, \{S_{Ea}\})$$

where $setState(S_I, t)$ specifies that the state of the SUT should be set to S_I at a time before t. and the event E should be generated and sent to IUT at time t. $earlyEvent(Event(E, \{IUT\}), t)$ specifies that the event or input $Event(E, \{IUT\})$ is sent to the SUT at time t. The expected behavior in that case is S_{Eb} or S_{Ea} depending on whether t is before or after t_0.

C. High input frequency

This pattern tests the behavior of the system when the input frequency is high and higher than what the system or module can handle given the resources and processing power. These tests are specified generically and since the frequency can be dependent on what platform and hardware the SUT is running on, RobusTest will increase the frequency of the input gradually until it comes to a state where the SUT can not handle the work load. At that point the SUT is expected to behave in accordance with the specification without crashing. To specify a property for this pattern the following factors are to be specified: IUT, S_I, f, S_E.

Test cases for this pattern start with setting the initial state S_I to the SUT and then generating inputs and events with higher frequency than f to check the behavior of the SUT in that case.

D. Lost events

This pattern discusses robustness issues that occur when an event is expected but is missing. Although this pattern is not directly a timing pattern it usually has a close correlation to the timing issues as seen in the first two patterns discussed above. The following parameters are to be specified in order to generate test cases for this pattern: E, S_I, E_F, S_E.

The test cases in this pattern aim to test the robustness of the SUT when an important event is lost or ignored. In order to simulate this situation the event E is not created or created in an erroneous manner in the test cases.

E. High output frequency

This pattern discusses the robustness issues resulted from high output frequency of a module and its consequences for the whole system or other systems. In the same manner as high input frequency to specify a property for this pattern the following factors are to be specified: IUT, S_I, f, S_E.

The high input from one module or unit can lead to missing events and messages or overloading other part of the SUT that might lead to robustness issues. Since the framework focuses mostly on black box testing it is not always possible to generate tests for this pattern. Although if the structure of the SUT allows this, it can be simulated by limiting the output channels of the system or the input of the receiving unit.

F. Input before or during startup, after or during shut down

The focus of this pattern is to test the behavior of the system towards inputs received during startup or shut down. The test cases for this pattern can be generated in the same way as for input with unexpected timing. The main focus here is though to check whether inputs during startup and shut down can change the state of the SUT in a way that causes irregularities after the SUT has started properly.

G. Error recovery delays

This pattern focuses on the state where the SUT is recovering from an error or has degraded functionality. In the same way as the previous pattern inputs received during error recovery needs to be handled in a specific way. Although the SUT is running during these phase but the functionality is degraded and the transmitting parts and modules need to be aware of that and the received input needs to be handled properly according to the requirement specification for the SUT.

IV. Conclusion

This paper has discussed a framework called RobusTest for testing robustness properties of software systems. In the current version the framework mostly focuses on timing issues that lead to robustness vulnerabilities. Testing is done using properties that are written to specify an expected behavior from the system. These properties are then used to generate test cases automatically. The properties are even used to analyze the results from executing the test cases on the system and is in this way used as an oracle for the behavior of the system.

In this paper, seven properties from ROAST, which is a framework for specifying and testing robustness properties in software, are introduced. RobusTest is a part of ROAST with focus on the testing part. In the current study, the patterns with focus on timing issues were presented and the properties extracted from each pattern were discussed in more detail. After writing properties based on the patterns there is a clear link through those properties from the requirements elicited and specified by ROAST and the test cases generated by RobusTest.

V. Current and Future Work

Given the structure presented in this paper, the framework is being built for Java as an extension of JUnit. However, this framework can be implemented in any programming language and the Java framework is a case study for proof of concept. The idea is to extract commonalities for requirements in the same patterns and wrap them in RobusTest for testing those requirements. In this manner, there will be a common interface with built in functionality such as generation of test cases and automated CRASH oracle that can be used to test a specific type of requirement. This JUnit extension is then to be tested initially for protocol testing in a communication protocol with timing restrictions.

This paper aimed to present the idea and structure of the framework. However, a small evaluation of the concept was performed by testing parts of the framework on two simple programs. The generated test cases were able to identify the faults that were injected in the programs. However, since this evaluation was very small compared to the size of the framework, it is not possible to draw any conclusion regarding the validity of RobusTest. A more thorough evaluation on two large open source systems is currently in progress and will be published in our future publications.

Another important next step is to look at other patterns in ROAST that are not currently included in RobusTest. Testing for input with invalid value and testing unexpected conditions in the execution environment are to be added to RobusTest in order to have a complete structure and a clear link from the requirements in those patterns and the test cases.

REFERENCES

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

[2] A. Shahrokni and R. Feldt, "Towards a Framework for Specifying Software Robustness Requirements Based on Patterns," *Requirements Engineering: Foundation for Software Quality*, pp. 79–84, 2010.

[3] J. DeVale, P. Koopman, and D. Guttendorf, "The Ballista software robustness testing service," in *Testing Computer Software Conference*, 1999.

[4] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software-Practice & Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[5] A. K. Ghosh, M. Schmid, and V. Shah, "Testing the robustness of Windows NT software," in *Proceedings of the 9th International Symposium on Software Reliability Engineering, 4-7 Nov. 1998*, ser. Proceedings of the 9th International Symposium on Software Reliability Engineering (Cat. No.98TB100257). Los Alamitos, CA, USA: IEEE Computer Society, 1998, pp. 231–235.

[6] M. Dix and H. D. Hofmann, "Automated software robustness testing - static and adaptive test case design methods," in *Proceedings of the 28th Euromicro Conference, 4-6 Sept. 2002*, ser. Proceedings of the 28th Euromicro Conference. Los Alamitos, CA, USA: IEEE Computer Society, 2002, pp. 62–66.

[7] J. Fernandez, L. Mounier, and C. Pachon, "A model-based approach for robustness testing," *Testing of Communicating Systems*, pp. 333–348, 2005.

[8] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proceedings of the 16th Symposium on Reliable Distributed Systems, 1997.*, 1997, pp. 72–79.

[9] D. Clarke and I. Lee, "Automatic generation of tests for timing constraints from requirements," in *words*. Published by the IEEE Computer Society, 1997, p. 199.

[10] G. Fink and M. Bishop, "Property-based testing: a new approach to testing for assurance," *SIGSOFT Softw. Eng. Notes*, vol. 22, pp. 74–80, July 1997.

[11] G. Fink and K. Levitt, "Property-based testing of privileged programs," in *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*. IEEE, 1994, pp. 154–163.

[12] G. Fink, C. Ko, M. Archer, and K. Levitt, "Towards a property-based testing environment with applications to security-critical software," in *Proceedings of the 4th Irvine Software Symposium*, vol. 39, 1994, p. 48.

[13] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, pp. 268–279, September 2000.

[14] J. Hughes, U. Norell, and J. Sautret, "Using temporal relations to specify and test an instant messaging server," in *Proceedings of the 5th Workshop on Automation of Software Test*, ser. AST '10. New York, NY, USA: ACM, 2010, pp. 95–102.

Simulated Injection of Radiation-Induced Logic Faults in FPGAs

Cinzia Bernardeschi, Luca Cassano, Andrea Domenici
Department of Information Engineering
University of Pisa, Italy
Pisa, Italy
first_name.last_name@ing.unipi.it

Giancarlo Gennaro, Mario Pasquariello
Intecs S.p.A.
Pisa, Italy
first_name.last_name@intecs.it

***Abstract*—SRAM-FPGA systems are simulated with a model based on the Stochastic Activity Networks (SAN) formalism. Faults are injected into the model and their propagation is traced to the output pins using a four-valued logic that enables faulty logical signals to be tagged and recognized without recurring to a comparison with the expected output values. Input vectors are generated probabilistically based on assumed signal probabilities. By this procedure it is possible to obtain a statistical assessment of the observability of different faults for the generated inputs. The analysis of a 2-out-of-2 voter is shown as a case study.**

***Keywords*-SRAM-FPGA; Simulation; Single Event Upset; Single Event Transient; Stochastic Activity Networks**

I. Introduction

In the last decade SRAM-FPGAs played a very important role in the market of silicon devices, thanks to the low cost and relatively good performance. In the last years FPGAs have increasingly been employed also in safety-related applications such as railway signaling [1], radar systems for automotive applications [2] and wireless sensor networks for aerospace [3].

The industrial use of electronic devices in safety-critical systems imposes a rigorous system design and the identification of hazardous failure modes. This is particularly true for programmable electronic devices, such as FPGAs, since the failure modes observable at the boundary of the system strongly depend on the application implemented in the device.

Radiations in the atmosphere are responsible for introducing *Single Event Upsets* (SEU) and *Single Event Transients* (SET) in digital devices [4], [5]. SEUs have particularly adverse effects on FPGAs using SRAM technology, as they may alter a bit in the configuration memory, causing a permanent fault (correctable only with a reconfiguration of the device) [6]. SETs may temporarily alter the behaviour of user resources, such as flip-flops and multiplexers.

In this work, we present a simulation based fault injector for SRAM-FPGA systems that can be used for the analysis of radiation-induced logic faults. The FPGA is considered at the netlist level and SEUs and SETs affecting the logic resources of FPGAs are considered. The simulator is based on a model of SRAM-FPGA systems described with the Stochastic Activity Networks (SAN) formalism [7] and developed with the Möbius tool [8]. Faults are injected into the model and their propagation is traced to the output pins using a four-valued logic (along the lines of the D-calculus [9]) that enables faulty logical signals to be tagged and recognized. Input vectors are generated probabilistically based on assumed signal probabilities. For every generated test pattern (i.e., a sequence of input vectors), each possible fault in the adopted model is injected. By this procedure it is possible to obtain a statistical assessment of the observability of different faults for the given test patterns.

The remainder of this paper is organized as follows: Section II, briefly discusses the state of the art in the FPGA fault injection field; in Section III, the considered fault model is presented; Section IV, shows the SAN formalism and the Möbius tool; in Section V, the model of FPGA-based systems and the fault injector are presented; in Section VI, the simulator engine, the available measures and an example of application are shown; Section VII, concludes the paper.

II. State of the Art

Fault injection is a widely used approach to evaluate the propagation of faults in digital devices. Fault injection techniques for SRAM-FPGA based systems can be divided into prototype-based [10], [11] and simulation-based [12], [13]. Prototype-based techniques have high performance and accuracy, but, since they are performed at the end of the design process, they make corrections expensive. Additionally they often depend on the particular vendor and model of the FPGA chip. Simulation-based techniques alleviate these problems offering the designer greater observability and controllability, but their accuracy may be limited by the assumptions on the system and fault model.

To the best of our knowledge, simulated fault injection for FPGAs at the netlist level has been proposed only in [12] and in [13], but, unlike our method, these tools are not entirely based on simulation, since they rely on an underlying prototype-based analysis. Further, with respect to both [12] and [13], the four-value logic allows us to recognize faulty signals without recurring either to a golden run or a golden copy of the system. Our choice of considering the system at the netlist level is due to the fact that at the register-transfer level (i.e., VHDL or Verilog description) faults in the hardware structure of the system can not be analysed. Moreover, the SAN model is quite general and allows different kind of analyses to be performed, such as failure probability computation [14].

III. Fault model

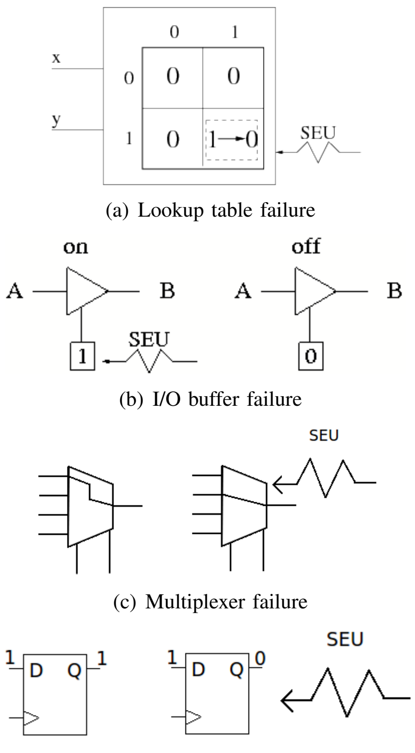
An FPGA is a prefabricated array of programmable blocks, interconnected by a programmable routing architecture and surrounded by programmable I/O blocks [15].

Programming an SRAM-FPGA device consists in downloading a programming code, called a *bitstream*, into its configuration memory. The bitstream determines the functionalities of logic blocks, the internal connections among logic blocks and the external connections among logic blocks and I/O pads. Interconnections are realized internally by routing switches and externally by *I/O buffers*. The most common programmable logic blocks are *lookup tables* (LUT), small memories whose contents are defined by configuration bits.

In this work the FPGA system is modelled at the netlist-level representation produced in the synthesis phase before the place and route. At this level, the elements visible in the model are I/O buffers, LUTs, flip-flops, and multiplexers. We consider both SEUs in the configuration memory of LUTs and I/O buffers and SETs in multiplexers and flip-flops

A SEU in the configuration memory of a LUT causes the alteration of the functionality performed by the LUT. Figure 1(a) shows a SEU causing a bit flip in the configuration bit associated to input `(1 1)`. In this case the logic function implemented by the LUT changes from an AND to a constant 0. I/O buffers are connecting resources placed at the input and output of the chip. Each buffer is opened/closed by a configuration bit. A SEU in the configuration bit of a buffer causes an undesired connection/disconnection between two wires, as shown in Figure 1(b).

A SET in a multiplexer causes the temporary selection of a wrong signal, as shown in Figure 1(c). Finally a SET in a memory element, such as a flip-flop (see Figure 1(d)), causes the storage of a wrong value, until a new value is written in the flip-flop.



(a) Lookup table failure

(b) I/O buffer failure

(c) Multiplexer failure

(d) Flip-flop failure

Figure 1. Failure modes of various resources of the FPGA chip.

IV. The SAN Formalism

SANs [7] are an extension of Petri Nets (PN). SANs are directed graphs with four disjoint sets of nodes: *places*, *input gates*, *output gates*, and *activities*. The topology of a SAN is defined by its input and output gates and by two functions that map input gates to activities and pairs (*activity*, *case*) (see below) to output gates, respectively. Each input (output) gate has a set of *input* (*output*) places.

The activities replace and extend the *transitions* of the PN formalism. Any activity may have mutually exclusive outcomes, called *cases*, chosen probabilistically according to the *case distribution* of the activity.

As in PNs, the state of a SAN is defined by its *marking*. The marking of each place is a non-negative integer (called the *number of tokens* of the place).

SANs enable the user to specify any desired enabling condition and firing rule for each activity. This is accomplished by associating an *enabling predicate* and an *input function* to each input gate, and an *output function* to each output gate. The enabling predicate is a Boolean function of the marking of the gate's input places. The input and output functions compute the next marking of the input and output places, respectively, given their current marking.

Graphically, places are drawn as circles, input (output) gates as left-pointing (right-pointing) triangles, and activities as vertical bars. Cases are drawn as small circles on the right side of activities. Gates with default (standard PN) enabling predicates and firing rules are not shown.

A. The Möbius Tool

Möbius [8] is a popular software tool that provides a comprehensive framework for model-based evaluation of system dependability and performance.

SAN models can be composed by means of *Join* and *Rep* operators. Join is used to compose two or more SANs. Rep is a special case of Join, and is used to construct a model consisting of a number of replicas of a SAN. Models composed with Join and Rep interact via *place sharing*. Graphically, a composed model is represented as a tree whose nodes are either *atomic* models (i.e., simple SANs), or Join and Rep operators.

Properties of interest are specified with *reward functions*. A reward function specifies how to measure a property on the basis of the SAN marking. Measurements can be made at specific time instants, over periods of time, or when the system reaches a steady state. A desired confidence level is associated to each reward function. At the end of a simulation the Möbius tool is able to evaluate for each reward function whether the desired confidence level has been attained or not thus ensuring a high accuracy of the measurements.

V. MODELLING FPGAS WITH SANS

The FPGA model is split into a number of modules that interact through shared places [16]. Modules *System Manager*, *Input Vector*, *Combinatorial Logic*, and *Sequential Logic* describe the FPGA operation and module *Fault Injector* deals with faults.

The System Manager module orchestrates the activity of the other modules of the system according to the following steps: (i) a fault is injected; (ii) an *input vector*, i.e., an n-tuple of the input signal values, is applied to the input lines; (iii) the combinatorial part of the system is executed; (iv) a clock tick arrives and the sequential part of the system is executed. Steps (ii) through (iv) are repeated until all input vectors have been applied.

The Input Vector module applies an input vector to the input lines of the FPGA.

The Combinatorial Logic module models the combinatorial part of the system. The modelled components are lookup tables, multiplexers, and I/O buffers.

The Sequential Logic module models the flip-flops in the FPGA. Various types of flip-flops can be modelled.

The Fault Injector module is in charge of injecting faults into the netlist. For the purpose of this work, the fault injector injects a single permanent fault into the configuration memory of LUTs and I/O buffers or a single transient fault in the user resources (flip-flops or multiplexers). The fault is injected at the beginning of the simulation. Faults are exhaustively injected in the system one at a time.

Combinatorial and sequential elements are modelled by a SAN model, called `Generic_Component` (see Figure 2(a)). Places `spA` and `spB` are used to control the execution of a component. The output gate `OG0` implements the functionality of the component. When the `execute` activity of a component completes, the function specified in gate `OG0` is executed, and a token is added to `spB`.

Three shared places (`input_lines`, `output_lines`, and `internal_lines`) encode the value of the signals on the input, output, and internal connections of the FPGA. The shared place `faults` keeps track of the faults injected in the system. Components behave correctly or faulty according to the content of place `faults`.

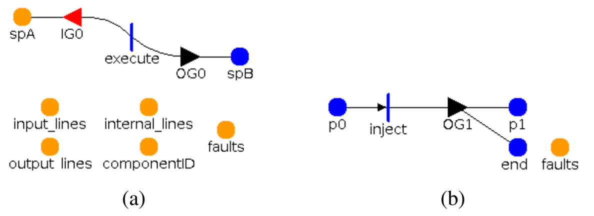


Figure 2. Generic_Component (a) and Fault_Injector (b) module.

The SAN model of the Fault Injector is shown in Figure 2(b). Places `p0` and `p1` are used to control the execution of the fault injector. Place `faults` is shared with the combinatorial logic module. Place `faults` is an array of *C* Boolean values, where C is the number of configuration bits associated to LUTs and I/O buffers, plus the number of flip-flops and multiplexers. In particular `faults[i]` equals 1 if the i-th configuration bit is faulty or the associated user resource is faulty. The output gate `OG1` implements the fault injection function resetting the element of `faults` associated to the previously injected fault and setting the element affected by the new fault. When the `inject` activity completes the function specified in gate `OG1` is executed, and a token is added to `p1`. When every possible faults have been injected and the associated simulation runs have been performed, a token is placed into `end`.

The logical connections are specified in a *connectivity matrix*, a data structure accessed by the input and output functions of the model. This way, the logical connections are not hardwired in the SAN models, and can be set up starting from netlist EDIF file generated by CAD tools, such as the Xilinx ISE tool, on the basis of the specification of the FPGA-based system.

VI. THE SIMULATOR

The simulator executes the previously discussed model of FPGA-based systems on a four-valued logic that enables faulty logical signals to be tagged as such and followed along their propagation path. In this logic we distinguish correct values from faulty ones (as in D-Calculus [9]).

Correct and faulty Boolean values are named zero correct (0_c), one correct (1_c), zero faulty (0_f) and one faulty (1_f). More precisely, let $B = \{0, 1\}$ and $D = \{0_c, 0_f, 1_c, 1_f\}$, where B is the set of standard Boolean values and D is the domain of the four-valued logic. Then we establish a correspondence between B and D by the following mappings:
$\phi : D \to B$, such that $\phi(0_c) = \phi(0_f) = 0$ and $\phi(1_c) = \phi(1_f) = 1$, is a *projection* function that translates values of D to values in B ignoring the faulty/correct annotation.
$\chi : D \to B$, such that $\chi(0_c) = 0$, $\chi(1_c) = 1$, $\chi(0_f) = 1$ and $\chi(1_f) = 0$, is a *corrective* projection that replaces a faulty value with its complemented Boolean value (i.e. it extracts a correct value from a faulty one).

Then we define *tracking* functions for components. These

x	y	$\wedge^*$
0_c	-	0_c
-	0_c	0_c
1_c	1_c	1_c
1_c	0_f	0_f
1_c	1_f	1_f
0_f	1_c	0_f
1_f	1_c	1_f
0_f	0_f	0_f
0_f	1_f	0_c
1_f	0_f	0_c
1_f	1_f	1_f

x	y	$\vee^*$
1_c	-	1_c
-	1_c	1_c
0_c	0_c	0_c
0_c	0_f	0_f
0_c	1_f	1_f
0_f	0_c	0_f
1_f	0_c	1_f
0_f	0_f	0_f
0_f	1_f	1_c
1_f	0_f	1_c
1_f	1_f	1_f

x	$\neg^*$
0_c	1_c
1_c	0_c
0_f	1_f
1_f	0_f

function q^*		
D	Q_{prev}	Q
-	0_c	0_c
-	1_c	1_c
-	0_f	0_f
-	1_f	1_f

function q_c^*		
D	Q_{prev}	Q
0_c	-	0_c
1_c	-	1_c
0_f	-	0_f
1_f	-	1_f

Table I
FOUR-VALUED TRUTH TABLES FOR AND, OR, NOT, AND D EDGE-TRIGGERED FLIP-FLOP.

functions trace the propagation of values through components. Each non-faulty component implements a Boolean function $f : B^n \rightarrow B$. For such function, its tracking function $f^* : D^n \rightarrow D$ extends the semantics of f to the four-valued domain D. For a given n-tuple of inputs $(d_1, \cdots, d_n)$ in D^n, this function evaluates f both with the projection of $(d_1, \cdots, d_n)$ to B^n (i.e., $(\phi(d_1), \cdots, \phi(d_n))$) and with the corrective projection of $(d_1, \cdots, d_n)$ to B^n (i.e., $(\chi(d_1), \cdots, \chi(d_n))$). This amounts to applying f to the actual inputs and to the input that would have been applied in absence of faults. Function f^* compares the two results. If they are equal, the result is $f(\phi(d_1), \cdots, \phi(d_n))$ tagged as a correct value, otherwise the result is $f(\phi(d_1), \cdots, \phi(d_n))$ tagged as faulty.

In particular, we define the four-valued logical operators $\wedge^*$, $\vee^*$ and $\neg^*$ as the tracking functions of the corresponding Boolean operators. The semantics of these operators is given by truth tables (see Table I). We may notice that for the $\wedge^*$ operator, a 0_c on one input masks any faulty value on the other input; similarly for the $\vee^*$ operator a 1_c masks any faulty value on the other input.

We defined the tracking function for the components of the netlist: I/O buffers, LUTs, multiplexers, and flip-flops. Flip-flops are modelled with two functions: the first one models the behaviour of the flip-flop in the presence of a clock rising edge (called q_c^*), the other (q^*) models the behavior of the flip-flop during the inactive period. For example, the functions for a standard D-Edge Triggered flip-flop are shown in Table I. We may notice that the output (correct or faulty) is unchanged in absence of a rising edge, while it follows the input when a rising edge occurs.

We now show how to model the generation of faulty values by faulty components and the propagation of values through faulty components. Given a Boolean function $f : B^n \rightarrow B$ implemented by a logic component, for each possible fault i of the component we define a *faulty* function $\hat{f}_i : B^n \rightarrow B$ that describes the behaviour of the component in presence of that fault. This behaviour may be given in the form of truth table or as an expression. For simplicity, in the following we will drop the subscript.

Then, the tracking function $\hat{f}^*$ of a faulty function $\hat{f}$ compares the output of the faulty component with possibly faulty inputs to the output of the correct component with correct inputs. If the two are equal, the result is taken as correct. Otherwise it is tagged as faulty.

For example, given a two-input LUT implementing the AND function, let us assume that a fault occurs in the configuration bit associated with the input $x = 1$ and $y = 1$ (Figure 1(a)). When the input is $(1, 1)$, the output of the LUT is 0 instead of 1 (the output of the faulty LUT is always 0).

Function f_{LUT} describes the behaviour in absence of faults: $f_{LUT}(d_1, d_2) = d_1 \wedge d_2$, whereas function $\hat{f}_{LUT}$ represents the behaviour of the faulty LUT: $\hat{f}_{LUT} = 0$ if $d_1 = 1$ and $d_2 = 1$, $\hat{f}_{LUT} = f(d_1, d_2)$ otherwise.

Table II shows three cases for the tracking function of the faulty LUT: in the first case, two correct inputs activate the fault and generate a faulty output; in the second case the correct input 1_c and the faulty input 0_f do not activate the fault. However, the resulting output differs from the output that should have been produced with 1 and 1, and the faulty value is propagated. In the third case the correct input 1_c and the faulty input 1_f activates the fault but the resulting output equals the output that should have been produced with 1 and 0. We notice that in the course of simulation, the tracking functions can be calculated off-line and synthesized as truth tables before starting the simulation.

d_1	d_2	$\hat{f}(\phi(d_1), \phi(d_2))$	$f(\chi(d_1), \chi(d_2))$	$\hat{f}^*$
1_c	1_c	0	1	0_f
1_c	0_f	0	1	0_f
1_c	1_f	0	0	0_c

Table II
AN EXCERPT OF THE FAULTY LUT TRUTH TABLE

Using this four-valued logic we are able to trace the propagation of faults and to determine whether they reach the output, and, if not, to find which components mask or propagate the fault. This four-valued logic allows the observability of faults to be measured (a comparison of the actual output values with the expected ones is not necessary).

A. Simulation and Measurements

The configurable parameters of our simulations are the number of simulated clock cycles N and the signal probability of input signals SP_i, i.e. the probability of the signal to be 1 at a given time [17].

In order to measure the fault observability of the system under analysis we perform multiple simulation runs of the system. Each simulation run is structured in the following steps, graphically represented by Figure 3:

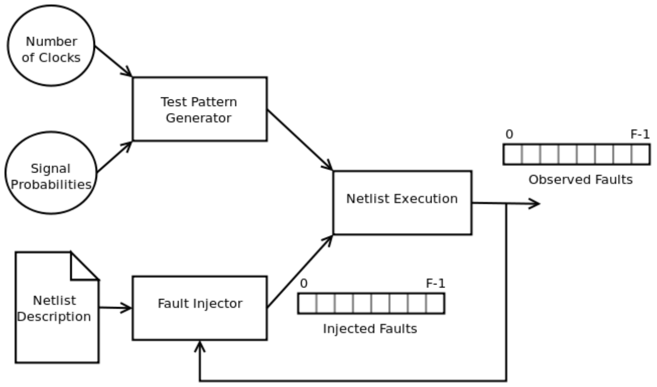


Figure 3. Functional blocks of the simulator.

1) A test pattern is stochastically generated according to given input signal probabilities and a number of clocks.
2) A fault is injected in the system.
3) The netlist is executed until the maximum number of clock cycles is reached. The following reward function detects system failures:

```
if(System_Manager->clock->Mark()==1){
   for(int i=0; i<N_out; ++i)
      if(System_Manager->output_lines->
         Index(i)->Mark() == 1f ||
         System_Manager->output_lines->
         Index(i)->Mark() == 0f)
             return 1;
  return 0; }
```

4) If more faults have to be injected, the current fault is removed and the simulation re-starts from step 2, otherwise simulation terminates.

Data that can be obtained with our analysis are the list of observed faults for each generated test pattern, and the total number of observed faults using the generated test patterns. From these data we can compute a quality factor, called *total observability*, of the set of test patterns, defined as the ratio of observed faults to the total number of injected faults.

The above shown reward function allows the analysis of the observability of faults at the output of the system. Other analyses can be performed: the behavior of any internal signal can be observed and, if a certain fault has been activated and it has not been observed at the output, we can find where the fault has been masked. Moreover, we can model different fault hypotheses, such as multiple faults, or faults confined to a certain area of the device, simply modifying the initialization of the fault injector module.

B. An Example

In order to analyse the applicability of our method we considered as a simple case study an 8-bit 2-out-of-2 voter.

The behaviour of the system is the following:

- After a $0 \rightarrow 1$ transition of Data_Valid, the circuit starts reading serially 8 bits from Stream_A and Stream_B.

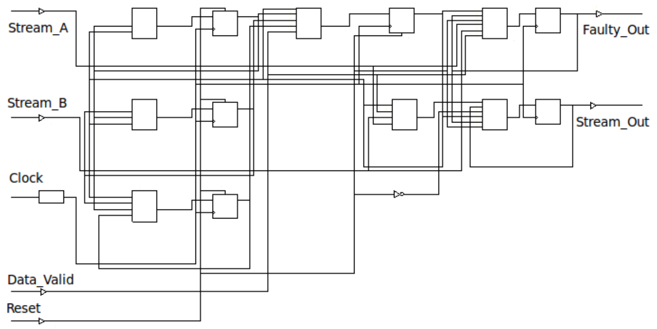


Figure 4. The netlist of the 8 bit 2-out-of-2 voter.

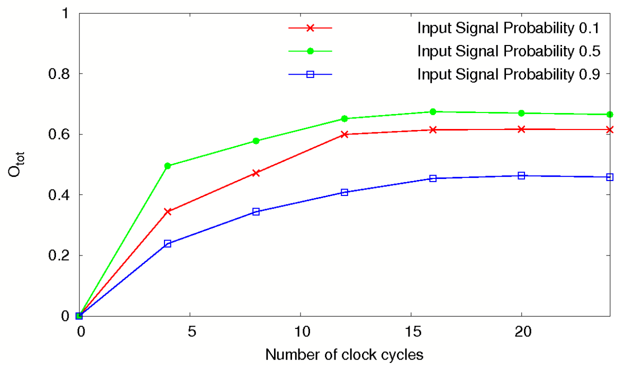


Figure 5. Total observability vs. test pattern length.

- If Stream_A and Stream_B are equal, Stream_Out follows Stream_A and Faulty_Out is 0.
- If Stream_A and Stream_B are different for at least one bit, Stream_Out is set to 0 and Faulty_Out to 1 for the rest of the byte.

We synthesised the system for the Xilinx Virtex 6 device into a netlist with the Xilinx ISE tool. The resulting netlist, (Figure 4), has 4 input signals, 2 output signals, 6 I/O buffers, 8 LUTs, and 6 flip-flops. We then used a parser from EDIF to our specification language to instantiate the model.

In every simulation we set the signal probability of the four input signals of the system to the same value.

In a first scenario we calculated the total observability of the system for $SP = 0.1, SP = 0.5$ and $SP = 0.9$, varying the number of simulated clock cycles. The resulting total observability is shown in Figure 5.

In a second scenario we calculated the total observability of the system for $N = 4, N = 8$ and $N = 12$ clock cycles, varying the signal probabilities of the input signals. The resulting total observability is shown in Figure 6.

Each simulation run took from 0.3 to 0.5 seconds to be carried out. In order to reach a confidence level of 0.95 with a confidence interval of 0.1, we needed from 2000 to 3000 simulation runs. The complete analysis required a few minutes to be performed.

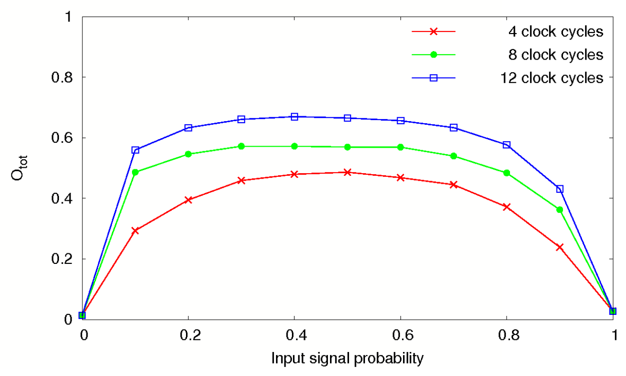


Figure 6. Total observability vs. input signal probability.

VII. Conclusions and Future Work

A simulation based fault injection tool for FPGA systems is shown. The FPGA system is modelled at netlist level. The considered fault model is fine-grained as the effect of SEUs affecting any configuration bit of a LUT and an I/O buffer can be simulated, as well as the effects of SETs in flip-flops and multiplexers. In this work the fault injector has been used for fault observability analysis. These measures can be used for giving details on the places in the logic design where injected faults have been/have not been observed. This information, given as feedback to designers, allows them to increase the system observability by reworking the logic around these places, for example by adding test points for the diagnosis of faults. As future work we intend to analyse the observability of other types of faults, such as faults in the routing architecture. Moreover, we intend to implement the generation of selective test patterns for fault diagnosis.

References

[1] J. Borecky, P. Kubalik, and H. Kubatova, "Reliable Railway Station System Based on Regular Structure Implemented in FPGA," in *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD '09)*, 2009, pp. 348 –354.

[2] V. Winkler, J. Detlefsen, U. Siart, J. Buchler, and M. Wagner, "FPGA-based Signal Processing of an Automotive Radar Sensor," in *Proceedings of the First European Radar Conference (EURAD)*, 2004, pp. 245 –248.

[3] J. Henaut, D. Dragomirescu, and R. Plana, "FPGA Based High Date Rate Radio Interfaces for Aerospace Wireless Sensor Systems," in *Proceedings of the Fourth International Conference on Systems (ICONS '09)*, 2009, pp. 173 –178.

[4] R. Baumann, "Radiation-induced Soft Errors in Advanced Semiconductor Technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305 – 316, September 2005.

[5] G. Wirth, F. Kastensmidt, and I. Ribeiro, "Single Event Transients in Logic Circuits Load and Propagation Induced Pulse Broadening," *IEEE Transactions on Nuclear Science*, vol. 55, no. 6, pp. 2928 –2935, 2008.

[6] P. Graham, M. Caffrey, J. Zimmerman, D. E. Johnson, P. Sundararajan, and C. Patterson, "Consequences and Categories of SRAM FPGA Configuration SEUs," in *Proceedings of the 6th Military and Aerospace Applications of Programmable Logic Devices (MAPLD'03)*, September 2003, p. n.a.

[7] W. Sanders and J. Meyer, "Stochastic activity networks: formal definitions and concepts," in *Lectures on Formal Methods and Performance Analysis*, ser. Lecture Notes in Computer Science, E. Brinksma, H. Hermanns, and J. Katoen, Eds. Springer Berlin / Heidelberg, 2001, vol. 2090, pp. 315–343.

[8] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius modeling tool," in *9th Int. Workshop on Petri Nets and Performance Models*. Aachen, Germany: IEEE Computer Society Press, September 2001, pp. 241–250.

[9] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, no. 4, pp. 278 –291, July 1966.

[10] L. Sterpone and M. Violante, "A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 4, pp. 965 –970, 2007.

[11] E. Johnson, M. Wirthlin, and M. Caffrey, "Single-Event Upset Simulation on an FPGA," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, June 2002, pp. 68–73.

[12] M. Violante, L. Sterpone, M. Ceschia, D. Bortolato, P. Bernardi, M. Reorda, and A. Paccagnella, "Simulation-Based Analysis of SEU Effects in SRAM-Based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 6, pp. 3354 – 3359, December 2004.

[13] G. H. Wang Zhongming, Yao Zhibin and L. Min, "A Software Solution to Estimate the SEU-induced Soft Error Rate for Systems Implemented on SRAM-based FPGAs," *Journal of Semiconductors (Chinese Institute of Electronics)*, vol. 32, no. 5, pp. 1–7, May 2011.

[14] C. Bernardeschi, L. Cassano, and A. Domenici, "Failure Probability of SRAM-FPGA Systems with Stochastic Activity Networks," in *Proceedings of the 14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, April, pp. 293 – 296.

[15] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, pp. 135–253, February 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1454695.1454696

[16] C. Bernardeschi, L. Cassano, A. Domenici, and P. Masci, "A Tool for Signal Probability Analysis of FPGA-Based Systems," in *Proceedings of the 2nd International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*, 2011, in press.

[17] V. Saxena, F. Najm, and I. Hajj, "Estimation of State Line Statistics in Sequential Circuits," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 3, pp. 455–473, 2002.

Concurrent Engineering used to Implement Risk & Hazard Control

Gheorghe Florea
Societatea de Inginerie Sisteme – SIS S.A.
Bucharest, Romania
e-mail: gelu.florea@sis.ro

Luiza Ocheana
University “Politehnica” of Bucharest
Bucharest, Romania
e-mail: luiza.ocheana@sis.ro

***Abstract* — In the current modern and industry-based society, automation is the key to success. The technology has changed over the last decades towards full control systems. Requirement specifications for Safety Instrumented Systems (SIS) form the core of Risk and Hazard (RH) assessment. SIS are the most flexible and effective tools for guarding the plants. Despite the debate pro and against the integrated approach of Basic Process Control System (BPCS) and SIS, more than a safety system is needed to keep the process running even with diminished functionalities instead of shutdown the plant. Our new approach is based on how a new hierarchical decision level can complete the mission regarding safety when the control room is not functional or cannot act properly in a hazard situation. Layers of protection should be used in order to reduce the risk to an acceptable level. The key is RH control implemented as a superior hierarchical level of decision and intervention. Concurrent Engineering (CE) applied to process control is the approach that can help the designers to achieve this level and efficiently use the proposed system architecture. Basically, a remote Process Help Center will host not only the copy of the process control system but the strategy and algorithms (RH control) to accomplish the safety task and to keep the process running. The CE and simulation are basic approaches to build the functionalities of new systems.**

Keywords – SIS; Redundancy; Remote intervention; Simulation; Diagnostics; Hierarchical decision; Concurrent engineering; Risk and hazard assessment.

I. INTRODUCTION

Process control and optimization represent the current way for safer and more efficient industrial plants, while risk management represents the starting point for new control algorithms and strategies. There is a stringent need for enhancing plant operations at production management level, because plants often operate near criticality, meaning in conditions far from the ideal ones from the point of view of control and stability. Continuous process industries are usually very complex and difficult to model and keep under control. While plant personnel feel there is a tremendous need for better and more versatile simulation and modeling tools, no product on the market offers the features necessary for dealing with the uncertain nature of complex plants [1].

Safety is an important issue nowadays that receives an increasing amount of focus lately. The reasons are, unfortunately, the numerous accidents that occurred in industrial plants, which compel the industry to take a better look at current practices like process design, process control, risk analysis and control, risk assessment. Worldwide engineering organizations have developed standards for the engineering of process safety. IEC released two standards IEC 61508 aimed at the suppliers of process safety equipment and IEC 61511 aimed at the end users of process safety equipment. ISA S84.01 “Application of Safety Instrumented Systems for the Process Industry” includes all elements from sensors to final elements, including inputs, outputs, power supply, logic solvers and user interfaces [2].

Applying these standards we obtain reliable facilities, but still we do not solve the problem of continuity of the production process - the main goal in the economic competition. This paper presents a way to solve the problem of maintaining the continuity of the process by introducing a level of control for risk and hazard situations.

In this paper, we present the challenge of Safety and Security systems (Section II), the introduction of Risk and Hazard control as a new level of decision (Section III), simulation as the key for Risk and Hazard control (Section IV), technologies to be used (Section V), results and conclusions (Sections VI and VII).

II. SAFETY AND SECURITY (SS) – THE CHALLENGE

In order to obtain the required level of safety and security, we must take into account four important phases: analyze the needed level of SS for the plant, design, implementation and maintenance.

Stand-alone safety systems have been the traditional method of choice, meaning separate design and operation requirements for Basic Process Control Systems (BPCS) and Safety Instrumented Systems (SIS) [3]. Separate systems were developed for process control and safety with proprietary operator interfaces, engineering workstations, configuration tools, data and event historians, asset management, and network communications. This approach affects the costs of infrastructure acquisition, plant systems integration, control and instrumentation hardware, wiring, project execution, installation, and commissioning, as well as ongoing expenses such as training, spare parts procurement, and logistics contracts [4]. Until recently, users had little choice other than to use completely different systems for control and safety. “A war of words is raging in the process control industry over the “integration” of safety and control systems. It’s a debate that has been ongoing for years, but the recent introduction of new integrated systems by several process controls vendors has lately added fuel to the fire” [5].

Today, integrating safety and control has become a cost effective choice for manufacturers that could not justify a separate SIS in the past. As a process manufacturer, you need to perform rigorous Risk and Hazard (RH) analysis based on IEC 61511 or ANSI/ISA-84.00.01 safety standards to decide on the right level of protection required for your plants. You may do that by selecting a SIS that provides close integration with the software tools of your BPCS while still providing the required degree of separation. Figure 1 illustrates the three options.

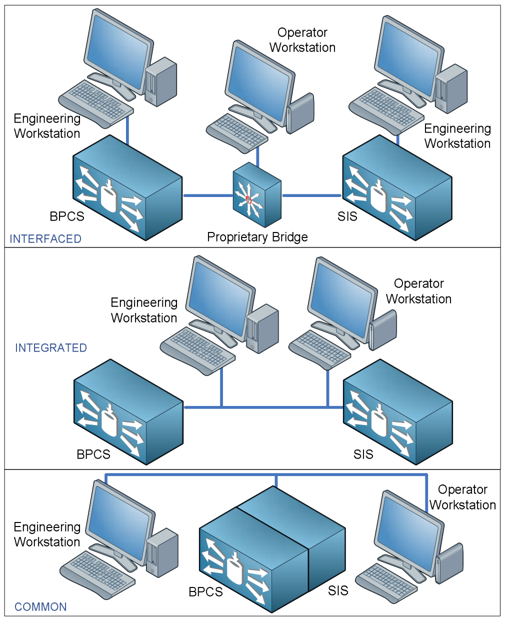


Figure 1 – SIS and BPCS Integration Levels

In the traditional sense, process safety refers to additional components that protect personnel and plant from injury, death and economic loss. However, many end users now recognize that the deployment of intelligent integrated safety solutions can directly improve process and personnel safety.

The entire issue of safety has direct influence upon the activity of the plant and therefore it must be integrated into the control system.

III. RH CONTROL – THE NEW LEVEL OF DECISION

According to process safety standards, the process risk has to be reduced to a tolerable level as set by the process owner [6]. The solution is to use multiple layers of protection, including the BPCS, alarms, Operator Intervention (OI), mechanical relief system and a SIS.

The BPCS is the lowest layer of protection and is responsible for the operation of the plant in normal conditions. If BPCS fails or is incapable of maintaining control, then, the second layer, OI, attempts to solve the problem. If the operator also cannot maintain control within the requested limits, then the SIS Layer must attempt to bring the plant in a safe condition [7]. If SIS also fails in restoring normal operation, then the hazard is imminent.

Risk is defined as the combination of the probability and the severity of a hazardous event, meaning how often it can appear and how severe are the consequences when it does. The best way to reduce risk in a manufacturing plant is to design safer processes. Unfortunately, it is impossible to eliminate all risks, so a manufacturer must agree on a level of risk that is considered tolerable. After identifying the hazards, a RH analysis must be performed to evaluate each risk situation.

The layers of protection and also the impact over the process are illustrated in Figure 2. On the left side, the layers of protection are listed; on the right side, the corresponding actions on the process are listed.

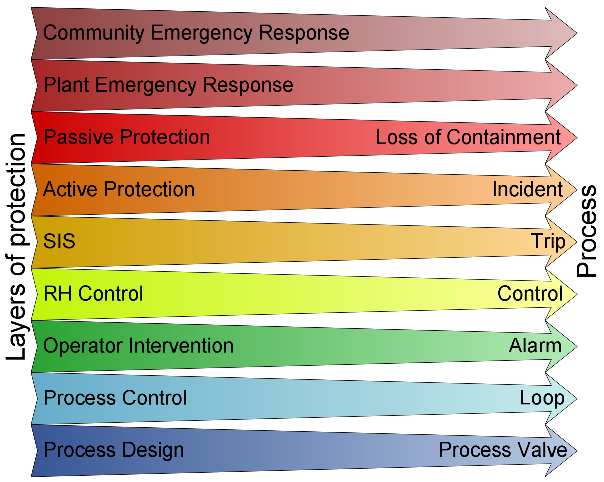
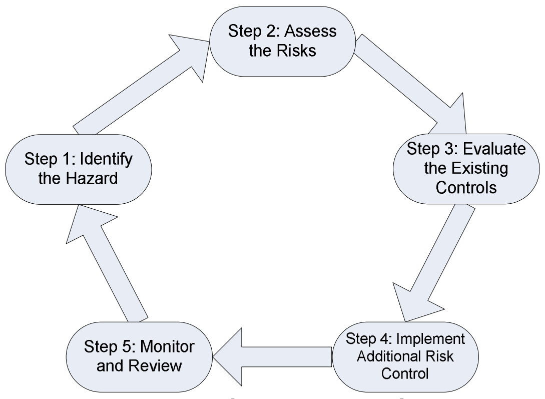


Figure 2 - Layers of protection and impact on process

Risk assessment procedure (detailed in Figure 3) is the first process in the risk management methodology to determine the extent of the potential threat and the risk associated with a system [8]. The procedure includes 5 important steps: (1) identify all possible hazard situations and (2) risks, (3) evaluation of the existing tools and strategies, (4) implementation of new ones if needed, and (5) the continuous monitor and evaluation of the behavior of the process.


Step 2: Assess the Risks
Step 1: Identify the Hazard
Step 3: Evaluate the Existing Controls
Step 5: Monitor and Review
Step 4: Implement Additional Risk Control
Figure 3 - Risk assessment procedure

BPCS, along with process alarms and facilities for manual intervention, provide the first level of protection and reduce the risk in a manufacturing facility. Additional protection measures are needed when a BPCS does not reduce the risk to a tolerable level. They include SIS along with hardware interlocks, relief valves, and containment dikes. Unfortunately, all the additional protection measures mentioned above can only help to safely shut down the plant.

We have proposed designed and implement a new level of decision: RH Control (Figure 4) to keep the plant running.

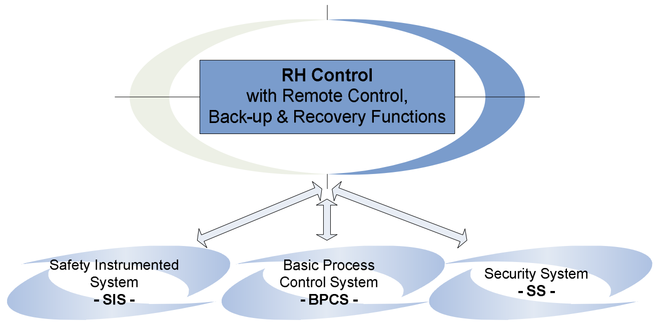

RH Control
with Remote Control,
Back-up & Recovery Functions
Safety Instrumented System - SIS -
Basic Process Control System - BPCS -
Security System - SS -

Figure 4 – System architecture

Better automation is a key aspect for improving industrial competitiveness [9]. Intelligent automation, at management levels in particular can play a major role regarding this aspect. The purpose of RH Control is to help with this improvement by building a new architecture and a distributed, generic decision support software system for near critical situation management in continuous process industries. In particular, assistance in terms of diagnosis and elaborating solutions is provided (directly to the plant's control system and/or to the staff) when certain situations are detected, i.e. situations suitable to be corrected, prevented or enhanced.

The focus is on new algorithms and strategies for the integration of different software components as well as on the system architecture itself. These software components include core, user interface and problem solving modules.

RH Control follows the conceptual structure of most distributed control systems that is a hierarchical and multilayered structure, similar to a pyramid. The complexity of the control mechanism increases for the higher layers. All the basic functionalities of the system are grouped into problem solving components that work in a cooperative way to find a solution to the plant problems or to optimize according to the plant objectives.

These applications include the following functionalities at different control layers:

- Strategies: Management of global objectives of the plant and their interrelation (management of maintenance operations, incident prevention, RH control, assessment of production costs in real time, loop tuning optimization, set-point deviation detection and alarm management)
- Tactics: Assistance through the problem lifespan, including process failure prevention, risk detection and diagnosis, plant-wide analysis, corrective actions, actions or recommendations for reestablishing effective control.
- Operations: Tasks such as filtering and validation of plant data, variable estimation, alarms analysis and optimization, intelligent alerting based on intuitive technologies and trend forecasting.

The main challenges at the beginning of a system configuration are: software architecture and reusability.

A. *Reusability*

The technical approach tries to provide reusability in the broadest sense using functional blocks. Object oriented technology can be one of the cornerstones of this approach [10]. Reusability can be achieved for any stage in the life cycle: from defining requirements and design to commissioning and maintenance. The approach is based on the availability of design template and reusable component implementation with few design compromises. These implementations are flexible enough to be adapted or modified to comply with the new requirements with little effort. The concepts of function block–based development and integration middleware provide the basis for reusability. RH Control will incorporate components for process control, risk analysis, optimization, etc.

The customized components will be integrated in a global architecture using real-time integration. This software, based on function block standard, will incorporate extensions to make possible for its use in real-time applications. This facilitates the easy reuse of components and even of the global application architecture because run-time components can be easily changed without affecting the behavior of others.

B. *Software architecture*

The software architecture is based on the Service–oriented architecture concept (SOA) [11]. In most applications, the infrastructure and the environment are very important security-related issues and it gets even more important if a SOA-based on Web Services has been chosen.

For this purpose, asymmetric cryptography will be used, implying a pair of two keys: public key and private key.

The benefits of this approach can be classified into two categories:

- From the user's point of view: the implementation addresses problems related to the global management of the plant while taking into account the interrelation of the strategic objectives, such as production, quality, maintenance, safety, efficiency and availability, as well as problems closer to the process control layer.
- From the systems integrator's point of view: the development of an open software architecture, based on the OPC standard and function blocks, will allow the construction of distributed intelligent control systems on top of the existing ones, with back-up functions.

IV. Simulation - the Key for an Effective RH Control

Future applications of simulation technology applied to process control will be driven by the advancing simulator capabilities. Many of them are the direct result of computing technology applied to certain activities with high return on investment: concurrent engineering, process fault detection, self testing capabilities for hardware and internet retrievable simulation models and tools.

- Advanced networking

Advances in network technology are allowing for faster data sharing between computers, parallel processing for simulating more complex models and linking the simulator with the real process. Three types of network interfacing applicable to simulation can be used:

- o Bus adapter and shared memory
- o Data broadcast network
- o Internet

• Intelligent I/O

Applied Dynamics International (ADI) developed and uses an intelligent input/output processor card to predict outputs and update the value more frequently than the update rate from the simulator, increasing speed for the next prediction

• Very High Speed Simulation

This approach is based on the development of digital hardware-in-the-loop simulations that allow simulation frame-times lower than 10 microseconds.

Simulation technology

There are many approaches to achieve good results in time. We will briefly present the most important of them.

• Integration algorithms

Integration algorithms are used to solve a function in the time domain, given the differential equation of the variable of interest. Runge-Kutta is probably the best known integration algorithm. A newer algorithm, named after its developers R. Bulirsch and J. Stoer is gaining popularity and may replace Runge-Kutta [12].

• Discrete-Event Simulation

Two types of discrete-event simulation tools are available: the state transition diagram editor and user / resource queuing tools.

State-transition diagram editors allow the user to model a process by the state the process is in and by the events that cause a transition from one state to another [13]. The use of state-transition diagrams allows the behavior of a process to be dependent on the state. A process simulator with a state – transition - diagram editor allows different dynamics to be assigned to different operational states of the same process. Figure 5 shows the classical states: start-up, nominal and shut-down but the RH state is added in order to maintain the system under control.

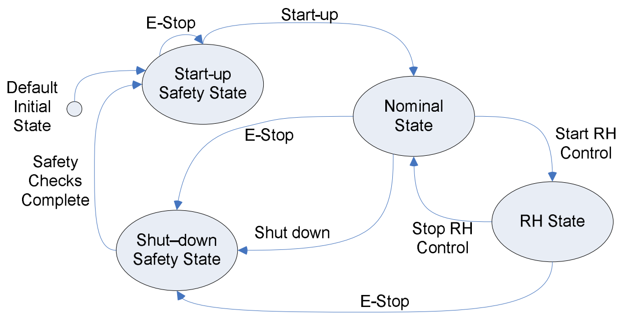


Figure 5 - Operational states

The user / resource analysis queuing system can be described as a collection of resources and the tasks using these resources [14]. The modeling tools allow resources to be allocated to tasks according to several prioritization strategies such as first-come-first-served, infinite servers, last-come-first-served, processor-sharing. System parameters such as response times, rate of use, queue populations and throughput rates can be assessed. Probability distributions and tasks attributes such as creating, terminating and delaying can be changed. This will be used later to implement the appropriate Distributed Control System (DCS) or Programmable Logic Controller (PLC) and Supervisory Control And Data Acquisition (SCADA) strategies to run on site or remote.

• System Identification

Data handling and processing power available today enables not only standard on-line identification techniques but also sophisticated, empirical model development methods that in the past were not feasible. Tools are available for today's simulators to help gather perturbation data from the process and develop empirical models that sometimes boast more fidelity than classical models. Although system identification theory has been around for a long time, only recently these theoretical tools become practicable because of the large amount of data processing required.

V. EMERGING TECHNOLOGIES

Emerging technologies analyzed helped us to establish the most important of them to be used in our project.

a) Concurrent Engineering (CE)

An activity that requires a high degree of effort from a design company, but not without a rewarding return on investment is CE. This design paradigm is based upon the principle that the process and the associated control strategy are designed in parallel before the process is built. Trade-off analysis is performed in advance, in order to prevent conflicting criteria of the two designs. Dynamic process simulators are combined with traditional static simulators to assess transient behavior and controllability of the process.

The CE approach is based on the following key elements:

- the system engineering process;
- a multidisciplinary team (process, control, safety and security, management, accounting, inventory)
- a collaborative platform, control environment and data & information distribution
- supporting tools and facilities.

The approach can evolve into an Integrated System Development based on cross functional System / Process Teams for all systems and services, and a System Engineering and Team to cover the system issues, performances, balance requirements.

By applying CE to plant design and installation, non-value added activities both in the upstream and downstream activities of the plant can be eliminated at the early stage of the design process, plant, operations and control. Plant wide controllability analysis in the conceptual design stage is an issue that has been raised by process industry [15].

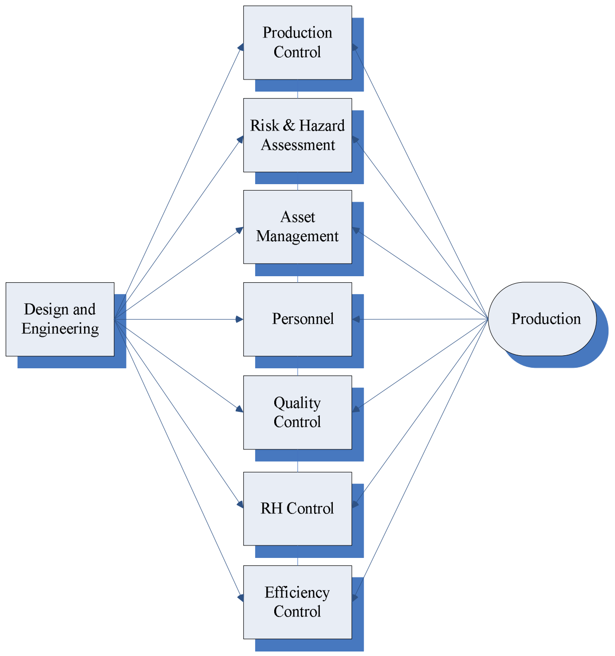


Figure 6 - Concurrent engineering apply to process control

The role of CE is obvious since it reflects that opportunities exist even at the conceptual design stage, to optimize the downstream operations including the capabilities to run the process instead of risks and hazards. This is against the conventional approach of the control as an add-on to process design after the flow sheet structure has already been determined.

There are a number of tools available for the design of process using CE including: simulation, process modeling, on – line identification, asset assessment, risk and hazard analysis. Including all this we can have a conceptual framework for the implementation of CE in process control.

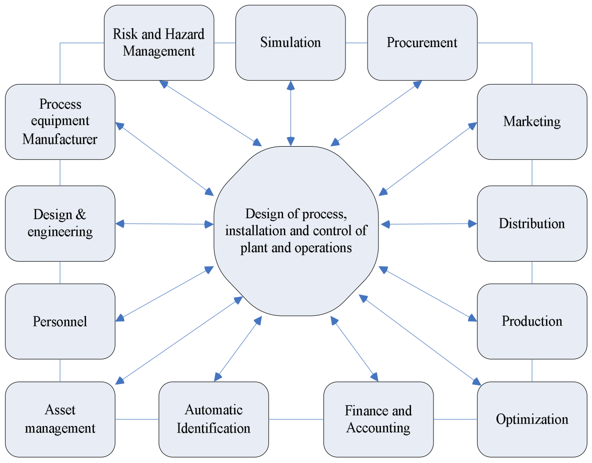


Figure 7 – Conceptual framework

b) Controller Testing

Using simulators to test control systems is an increasing trend in almost every industry. Simulator-based testing takes the control software development from the project critical path. Tests using simulators can be more comprehensive than a test using the actual process because the normal safety or process operational limits are not a concern, so the virtual test can exceed those limits, if necessary, to perform a more robust test. The networking options enable interfacing a simulator to a control system at a higher level in the architecture than in the past when individual wiring terminations were required.

c) On-line Diagnostics

Modern simulators offer the ability to detect faults in operating plants. A well tuned model of the plant runs in parallel with the plant, on-site or remote, comparing the model’s outputs with the real outputs. As shown in Figure 8, a difference between the two indicates a fault. Advanced fault-detection algorithms will lead the RH control or the supervisory engineers to the appropriate action.

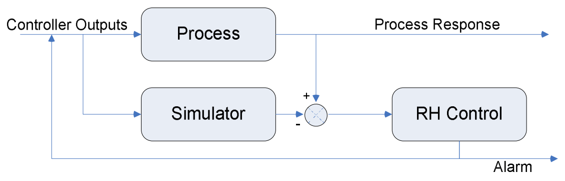


Figure 8 - Online diagnostics

d) Internet Applications

This technology offers today the capability to interconnect the on-site system with a remote control center (PH center) and to perform simulation, on-line identification [16], RH strategies, on-line tests and training, back-up and restoration. Operating remote from the site, the Process Help Center will host not only a copy of the process control system but the strategy and algorithms to fulfill the safety task and to keep the process running even in RH conditions.

VI. RESULTS

The new approach in process control system engineering, based on new algorithms, scalable and modular architectures and platforms, RH control, is industry independent. The capability of the systems to model and implement the 4 states, start-up, nominal, RH, shut-down, having 4 different strategies and the capability to change the state according to the functional parameters can be taken in consideration by CE. The diagnosis system, hosted remotely, will be continuously improved by gathering knowledge from various applications, based on identified problems, the solutions offered and their impact on the plant performance. The correlation factor between these different applications will influence future decisions. This way, the required period of time for solving a problem will be minimized, as well as the time that a plant needs to be shut down because of the instrumentation process control strategy.

Some of the expected results are an integrated exploitation of a collection of heterogeneous technologies for the prevention of anomalous situations related to the safety of an industrial complex and determining the suitability of function blocks and OPC based development for integrated control systems construction.

From the user's point of view, the accomplishment is that RH Control will allow the integration of the preventive and corrective aspects of safety, which were dealt, until this moment, in separate ways. Another advantage arises from being able to automatically take into account the constraints posed by the current plant situation and the ongoing maintenance operations.

The results achieved so far within the R&D project "Help Center and platform for remote diagnosis and remote intervention for the management of plants in hazardous situations – PH Center" will be used to develop and implement the hierarchically superior level for safety and security problems. The work carried out in the project establishes the baselines for a new architecture of process control taking into consideration the remote operation.

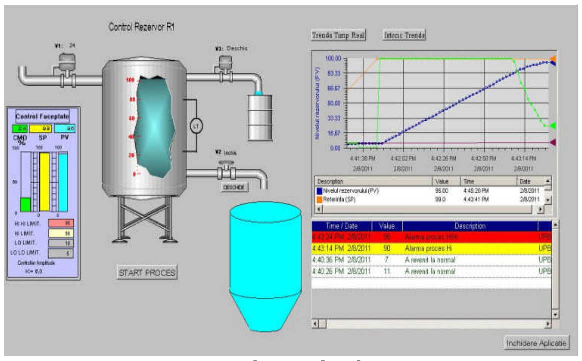


Figure 9 - The simulated process

In the meantime, the results achieved underlay the feasibility of the idea. This statement is based on two reasons:

- Two demo-applications have been designed according to real plant requirements with a large involvement of plant staff. At present, two applications are installed and under operation after a period of user validation and evaluation:

- a simulator for a simple process - controlling the level of liquid in a tank – Figure 9.

- a Building Management System (BMS) designed for a supermarket – Figure 10.

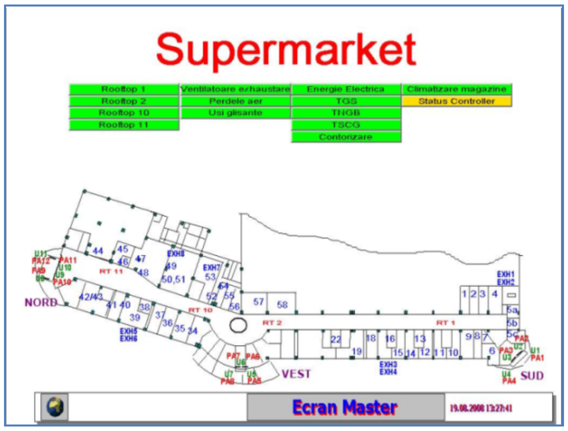


Figure 10 - Supermarket BMS – remote connection main screen

Also, we are currently working on including a new connection to the PH Center, a DCS control system (Experion - Honeywell) from LPG terminal, Midia, Navodari (Figure 11).

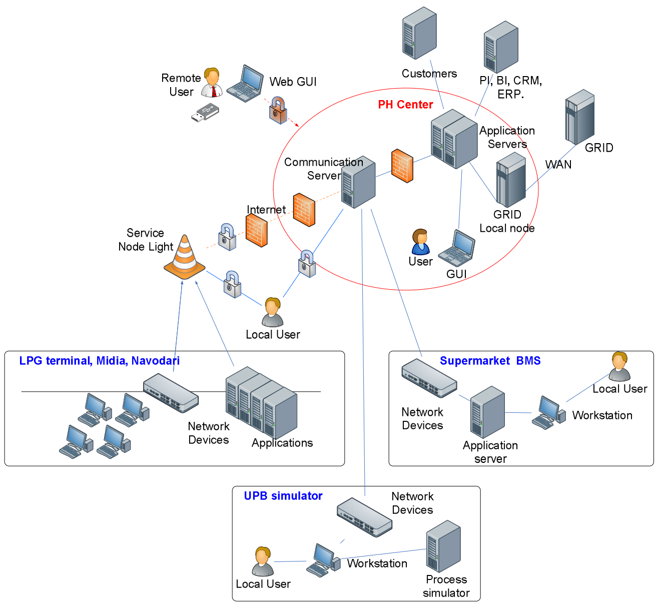


Figure 11 – PH Center connected to DCS

- The three generic products constructed within the project are truly reusable and can be exploitable components of other implementations.

This approach demonstrates that advanced control technology can be modularized, deployed and integrated with legacy control systems, progressing effectively towards complete automatic operation.

VII. Conclusions

In the past, SIS were strictly separated from the BPCS, mainly to segregate the safety and control functions and to have higher availability and reliability. Lately, many "integrated" control systems were deployed that have both BPCS and SIS systems in the same package

Hazard identification, risk assessment and control are on-going processes that involve a critical sequence of information gathering and also the application of a decision-making process. They assist in discovering what could possibly cause a major accident (hazard identification), how likely it is that a major accident would occur and the potential consequences (risk assessment) and what options exist for preventing and mitigating a major accident (control measures). The state of the art has no real and integrated solution. The work done by authors and the team has proposed:

- ✓ The new concept: Risk and Hazard Control;
- ✓ A new system architecture of process control;
- ✓ The guidelines and advantage of using CE to the design of process control system.

REFERENCES

[1] Gheorghe Florea, Luiza Ocheana, Radu Dobrescu, and Dan Popescu - Emerging Technologies - the base for the next goal of Process Control - Risk and Hazard Control, Proceedings of WSEAS International Conference, 2011, pp. 227 – 232.

[2] American Institute of Chemical Engineers - Guidelines for Safe and Reliable Instrumented Protective Systems, 2007.

[3] Asish Ghosh and Dave Woll - Business Issues Driving Safety System Integration, ARC White Paper, 2006.

[4] Ged Farnaby - Protect the plant. Leading edge trends in process control safety, InTech, June 2005.

[5] Wes Iversen - The Great Safety Debate, Automation World april 2007, pp. 30.

[6] David Hatch and Todd Stauffer - Operators on alert. Operator response, alarm standards, protection layers keys to safe plants. InTech, Cover Story, September 2009.

[7] Merry Spooner and Trevor MacDougall - Safety Instrumented Systems can they be integrated but separate?, ABB White Paper, 2011.

[8] Gary Stoneburner, Alice Goguen, and Alexis Feringa - Risk Management Guide for Information Technology Systems. Recommendations of the National Institute for Standards and Technology, 2002.

[9] Ricardo Sanz, Miguel Segarra, Angel de Antonio, and Idoia Alarcon - Plantwide Risk Management Using Distributed Objects, Proceedings of IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes 2, 2000, pp. 14 – 16.

[10] Michael Guttman and Jason R. Matthews - The Object Revolution, Wiley, New York, 1995.

[11] Stefan-Helmut Leitner and Wolfgang Mahnke. OPC UA – Service – oriented Architecture for Industrial Applications. ABB White Paper, 2006.

[12] Ruppel Francis and Wysor Wes - Mighty microprocessors boost process simulation, InTech, September 1997.

[13] David Harel - Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming vol.8, 1987, pp. 231 – 274.

[14] Christos Cassandras - Discrete Events Systems: Modeling and Performing Analysis, IFAC Best Control Engineering Textbook, 1999.

[15] Angappa Gunasekaran - Concurrent engineering: a competitive strategy for process industries, Journal of the Operational Research Society, Volume: 49, 1998, pp. 758-775.

[16] Mauro Coccoli and Antonio Boccalatte - Future Directions of Internet-based Control Systems, Journal of Computing and Information Technology, 2002, pp. 115–124.

Model Reconstruction: Mining Test Cases

Edith Werner and Jens Grabowski
Software Engineering for Distributed Systems Group
Institute for Computer Science,
University of Göttingen, Göttingen, Germany
{ewerner|grabowski}@cs.uni-goettingen.de

***Abstract*—System monitors need oracles to determine whether observed traces are acceptable. One method is to compare the observed traces to a formal model of the system. Unfortunately, such models are not always available — software may be developed without generating a formal model, or the implementation deviates from the original specification. In previous work, we have proposed a learning algorithm to construct a formal model of the software from its test cases, thereby providing a means to transform test cases for offline testing into an oracle for monitoring. In this paper, we refine our learning algorithm with a set of state-merging rules that help to exploit the test cases for additional information. Using the additional information mined from the test cases, models can be learned from smaller test suites.**

***Keywords*-Machine Learning, Reverse Engineering, Testing**

I. Introduction

Today, software systems are generally designed to be modular and reusable. A common scenario of a modular, reusable system is a web service, where simple services are accessed as needed by various clients and orchestrated into larger systems that can change at any moment. While the vision of ultimate flexibility is clearly attractive, there are also drawbacks, as the further usage of a module is difficult to anticipate. In this scenario, it may be advisable to monitor a system for some time after its deployment, to detect erroneous usage or hidden errors.

Monitors are used to observe the system and to assess the correctness of the observed behavior. To this end, monitors need oracles that accept or reject the observed behavior, e.g., a system model that accepts or rejects the observed traces of the monitored system. Unfortunately, the increasing usage of dynamic software development processes leads to less generation of formal models, as the specification of a formal model needs both time and expertise. Generating a formal model in retrospect for an already running system is even harder, as the real implementation often deviates from the original specification.

We propose a method for learning a system model from the system's test cases without probing the System Under Test (SUT) itself. When test cases are available, they often are more consistent to the system than any other model. Ideally, they take into account all of the system's possible reactions to a stimulus, thereby classifying the anticipated correct reactions as accepted behavior and the incorrect or unexpected reactions as rejected behavior. As the test cases are developed in parallel to the software, they provide a means to judge the correct behavior of the system. Also, test cases are generated at different levels of abstraction, e.g., for unit testing, integration testing, and system testing. By selecting the set of test cases to be used, the abstraction level of the generated model is influenced.

The basis of our approach is a learning algorithm, first introduced by Angluin [1], which learns a Deterministic Finite Automaton (DFA). To learn from test cases, we adapted the query mechanisms of the algorithm [2]. Experiments with our approach show that while a model can be learned this way, the algorithm only accepts simple traces as input, thereby losing additional information from the test cases, e.g., regarding branching, default behavior, or synchronization. We believe that exploitation of this additional information would enhance the learning algorithm.

In this paper, we propose a state-merging approach, termed *semantic state-merging*, which exploits the semantic properties of test cases in order to identify implicitly defined behavior. We first define a data structure, the *trace graph*, to store the available test cases. Then, we define merging rules for cyclic test cases and for test cases with default branches for the construction of the trace graph.

The remainder of this paper is structured as follows. Section II gives an overview on related work. In Section III, we introduce the foundations of our work in testing and machine learning. Section IV describes the trace graph and its construction. Based on this, Section V defines our approach to semantic state-merging on test cases. Subsequently, in Section VI, we give an overview on our experimental results. In Section VII, we conclude with a summary and an outlook.

II. Related Work

During the last years, a number of approaches have adapted Angluin's learning algorithm in combination with testing. Mainly, the approaches focus on the learning side of the problem and refine the properties of the generated model. Among the most recent adaptations are approaches to learning Mealy machines [3] and parameterized models [4], [5], [6], [7]. Some approaches can handle large or even infinite message alphabets [4] or potentially infinite state spaces [5]. In all those approaches, the learning algorithm generates test cases that are subsequently executed against the SUT, so that the System Under Test itself is the oracle for the acceptability of a given behavior.

Some approaches use outside guidance to improve the learning approach. The algorithm presented in [8] learns workflow petri nets from event logs and handles incomplete data by asking an external teacher. In [9], learning is used in a modeling approach. In this approach, a domain expert provides Message Sequence Charts representing desired and unwanted behavior.

Our approach differs from the above in two aspects. First, our aim is to generate a model for online monitoring. To this end, we need a model that is independent from the implementation itself. Therefore, we can neither use the implementation as an oracle nor learn from event traces generated by the implementation. Instead, we choose to learn from a test suite that was developed due to external criteria. Using a test suite also leads to the second difference of our approach. Where other approaches rely on unstructured data, a test suite provides relations between the distinct traces. We exploit those relations in order to enhance our learning procedure. Where other approaches address the learning side of the problem, our focus is actually on the structure of the teacher.

III. Foundations

In the following, the foundations of testing and on the learning of DFA are given.

A. Testing

A test case is itself a software program. It sends stimuli to the SUT and receives responses from the SUT. Depending on the responses, the test case may branch out, and a test case can contain cycles to test iterative behavior. To each path through the test case's control flow graph, a verdict is assigned. A common nomenclature is to use the verdict **pass** to mark an accepting test case and the verdict **fail** to mark a rejecting test case. An *accepting* test case is a test case where the reaction of the SUT conforms to the expectations of the tester. This can also be the case, when an erroneous input is correctly handled by the SUT. Accordingly, a *rejecting* test case is a test case where the reaction of the SUT violates its specification. Depending on the test specification, there may be additional verdicts, e.g., the Testing and Test Control Notation version 3 (TTCN-3) [10] extends the verdicts **pass** and **fail** with the additional verdicts **none**, **inconc**, and **error**: **none** denotes that no verdict is set; **inconc** indicates that a definite assessment of the observed reactions is not possible, e.g., due to race conditions on parallel components; and **error** marks the occurrence of an error in the test environment. During the execution of a test case, the verdict may be changed at different points. The overall assessment of a test case depends on the verdicts set along the execution trace, and is computed according to the rules of the test language. E.g., in TTCN-3, the overall verdict may only be downgraded, i.e., once an event was rated as **fail** the overall verdict may not go back to **pass**. For most SUTs, there is a collection of test cases, where each test case covers a certain behavioral aspect of the SUT. Such a collection of test cases for one SUT is called a *test suite*.

The main objective when constructing test cases for a software system is to assure that the specified properties are present in the SUT. To test against a formal specification, e.g., in the form of a DFA, test cases are derived from the model by traversing the model so that a certain coverage criterion is met, e.g., *state coverage* or *transition coverage*. State coverage means that every state of the model is visited by at least one test case. Transition coverage means that every transition of the model is visited by at least one test case. The largest possible coverage of a system model is *path coverage*, where every possible path in the software is traversed.

B. Learning a Finite Automaton Model from Test Cases

Our learning approach is based on a method proposed by Angluin [1]. The algorithm consists of the *teacher*, which is an oracle that knows the concept to be learned, and the *learner*, who discovers the concept. The learner successively discovers the states of an unknown target automaton by asking the teacher whether a given sequence of signals is acceptable to the target automaton. To this end, the teacher supports two types of queries. A *membership query* evaluates whether a single sequence of signals is a part of the model to be learned. An *equivalence query* establishes whether the current hypothesis model is equivalent to the model to be learned.

For learning from test cases, we need to redefine the two query types for test cases. The most important mechanism of the learning algorithm is the membership query, which determines the acceptability of a given behavior. In our case, the behavior of the software and thus of the target automaton is defined by the test cases. Since the test cases are our only source of knowledge, we assume that the test cases cover the complete behavior of the system. In consequence, we state that every behavior that is not explicitly allowed must be erroneous and therefore has to be rejected, i.e., *rejected* $\equiv$ $\neg$*accepted*. In consequence, we accept a sequence of signals if we can find a **pass** test case matching this sequence, and reject everything else.

The equivalence query establishes conformance between the hypothesis model and the target model. This is exactly what a test suite is designed for, therefore, we redefine the equivalence query as an execution of the test suite against the hypothesis model, where every test case in the test suite must reproduce its verdict. A detailed description of the learning algorithm can be found in [2], [11].

IV. Representing Test Cases

For the learning procedure, it is important that queries can be answered efficiently and correctly. Therefore, we need a representation of the test suite that is easy to search and provides a means to compactly store a large number of test cases. In the following, we define the *trace graph* as a data structure and describe its construction.

A. The Trace Graph

As described in Section III-A, in general, a test case is itself a piece of software and can therefore be represented as an automaton containing a number of event sequences. Usually, a test case distinguishes events received from the SUT, events sent to the SUT, and internal actions like value computation or setting verdicts. Each possible path through the test case must contain the setting of a verdict.

For the learning procedure, we only regard input and output events as the transitions in our target model and ignore internal actions except for the setting of verdicts. The verdicts are used to identify accepting test cases.

In general, every test case combines a number of traces, depending on the different execution possibilities. At the same time, a test suite contains a number of test cases, where different test cases may contain identical traces as they partly overlap. To present the test cases to the learning algorithm, we combine all traces from all test cases in the test suite into a single data structure, the *trace graph*, thereby eliminating duplicates and exploiting overlaps.

To enable an efficient search on the test cases, the trace graph is based on a labeled search tree, where all traces share the same starting state and traces with common prefixes share a path in the trace graph as long as their prefixes match. For the state-merging approach, the nodes in the trace graph are annotated with the verdicts. Cycles in the test cases are represented in the trace graph by routing the closing edges back to the starting node of the cycle. For better control, nodes where a cycle starts are also marked.

The trace graph forms the basic data structure for our semantic state-merging. The semantic state-merging methods depend on the information contained in the test cases, which in turn depends on the test language. To represent this, the trace graph can be extended to represent diverse structural information on the test cases by defining additional node labels. That way, information on the test cases will only affect the construction of the trace graph, but not the learning procedure that depends on its structure.

B. Constructing the Trace Graph

To construct the trace graph, we dissect the test cases into single traces and add them to the trace graph. Starting in the root of the trace graph, the signals in the trace to be added are matched to the node transitions in the trace graph as far as possible. We call this part of the trace the *common prefix*. The remainder of the new trace, the *postfix*, is then added to the last matched node. Algorithm 1 describes the procedure in pseudo code.

```
Data: A sequence of signals $w$
Start at the root node $n_0$ of the trace graph;
for all signal in $w$ do
    Get the first signal $b$ in $w$;
    if the current node has an outgoing edge marked $b$ then
        Move to the $b$-successor of $n$, which is $\delta(n, b)$;
        Remove the first signal from $w$;
    else
        // The signal is unknown at the current node
        Add $w$ as a new subgraph at the current node;
        return;
    end
end
```

Algorithm 1: Add a Trace to the Trace Graph

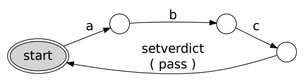
Cycles of the test case automaton need special treatment, as a cycle means that an edge loops back to an existing node. To this end, we separate the cyclic traces into three parts, a prefix leading into the cycle, the cycle itself and a postfix following the cycle. We then add the prefix and the cycle, whereby the last transition in the cycle is linked back to the beginning of the cycle. Finally, the postfix then is added to the trace graph.

V. Mining the Test Cases

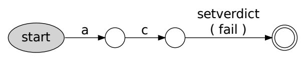
So far, the state-merging in the trace graph only means the combination of the test case automata, where traces are only merged as far as their prefixes match. The trace graph therefore exactly represents the test cases, but nothing more. In the following, we show two techniques to derive additional traces based on our knowledge of test cases.

A. Cycles and Non-Cycles

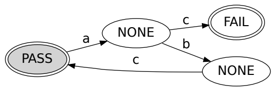
When testing a software system with repetitive behavior or a cyclic structure, the cycle has of course to be tested. However, usually it is sufficient to test the correct working of the cycle in one test case. In all other test cases the shortest possible path through the software is considered, which may mean that test cases execute only a part of a cycle or completely ignore a cycle. Depending on the test purpose, the existence of the cycle might not even be indicated in the test case. As long as the cycle itself is tested by another test case, the test coverage is not influenced. This approach results in shorter test cases, which means shorter execution time and thus faster testing. Furthermore, the readability of the test cases is increased. While the preselection of possible paths for cycles is appropriate for software testing, for machine learning it is desirable to have access to all possible paths of the software.



(a) A Test Case with a Cycle

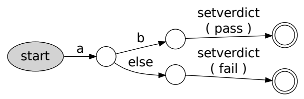


(b) A Test Case without a Cycle

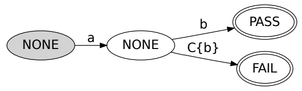


(c) Trace Graph Combining Both Test Cases

Figure 1. Precedence of Cyclic Behavior



(a) Test Case



(b) Trace Graph

Figure 2. Representing Default Branches

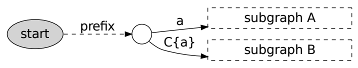


Figure 3. Generic Trace Graph with Default Branch

Consider the two test cases shown in Figures 1a and 1b. Although this is only a small example for demonstration purposes, the setting is quite typical. The test case shown in Figure 1a tests the positive case, that is, a repeated iteration of the three signals *a*, *b*, and *c*. The test case shown in Figure 1b tests for a negative case, namely what happens if the system receives the signal *c* too early. In the latter test case, the repetitive behavior is ignored, as it has been tested before and the test focus is on the error handling of the system. However, usually this behavior could also be observed at any other repetition of the cycle.

For the learning procedure, we would like to have all those possible failing traces, not only the one specified. We therefore define a precedence for cycles, which means that whenever a cycle has the same sequence of signals as a non-cyclic trace, the non-cyclic trace is integrated into the cycle. Figure 1c shows the trace graph combining the two test cases in Figures 1a and 1b. Besides the trace *a, c*, **setverdict(fail)** explicitly specified in Figure 1b, the trace graph also contains traces where the cycle is executed multiple times, *(a, b, c)*, a, c*, **setverdict(fail)**. With precedence of cycles, the test suite used as input to the learning algorithm can be more intuitive, as cycles only need to be specified once.

B. Default Behavior

Another common feature of test cases is the concentration on one test purpose. Usually, the main flow of the test purpose forms the test case, while unexpected reactions of the SUT are handled in a general, default way. Still, there may exist a test case that tests (a part of) this default behavior more explicitly.

Default branches usually occur when the focus of the test case is on a specific behavior, and all other possible inputs are ignored or classified as **fail**. Also, sometimes a test case only focuses on a part of the system, where not all possible signals are known. In such cases, the test case often contains a default branch, which classifies what is to be done on reading anything but what was specified.

For our application, this poses two challenges. The first challenge is in the learning procedure. For the different queries, we need to have as many explicitly classified traces as possible, but at the same time we do not want to blow up the size of the test suite. The second challenge is in the construction of the trace graph. When adding all different traces into one combined structure, the implicit context of what is “default” in the local test case is lost. Also, sometimes another test case uses the same default, adds more specific behavior in the range of the default, or defines a new default which slightly differs. We therefore need a method of preserving the local concept of “default” in the test cases and a method of combining different defaults in the trace graph.

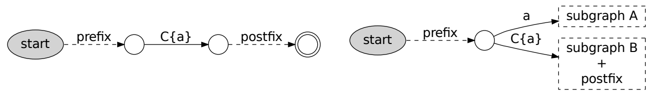
Consider a typical default situation, like a **default** statement in a **switch-case** environment. The **default** collects all cases that are not explicitly handled beforehand. As branching on alternatives splits the control flow in a program, each of the branches belongs to a different trace. Therefore, when taking the traces one by one, the context of the default is not clear. To preserve this context, instead of **default** we record the absolute complementary of the set of other alternatives, which is $\complement\{a, b\}$. A *complementary set* is a set that contains everything but the specified elements. Figure 2 shows a test case with defaults (Figure 2a) and its representation as a trace graph using the complementary set notation (Figure 2b). The branch marked with $\complement\{a\}$ represents every branch not marked with *a*.

Figure 3 shows a trace graph with a default branch in a general way. There are some arbitrary transitions leading to the default (marked with *prefix*), the default branching itself with an edge marked *a* and an edge marked $\complement\{a\}$ (”everything but a”), and the arbitrary subgraphs of *a* and $\complement\{a\}$.

When adding a trace with a matching prefix to this trace graph, the signal s following the prefix can be matched to the trace graph according to one of the following three cases.

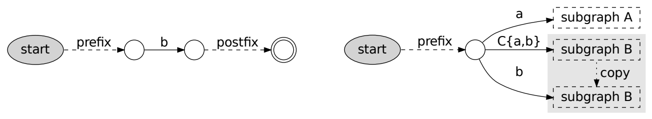
- *Exact Match:* s matches one of the branches of the trace graph, i.e., if s is a complementary set, it is identical to the complementary set in the trace graph.
- *Subset:* s matches one signal (or a subset of signals) of the complementary set in the trace graph.
- *Overlap:* s is a complementary set, and overlaps the complementary set in the trace graph.

The first and simplest case is the *exact match*, where a trace with a matching complementary set is added. As the complementary sets are identical, it suffices to add

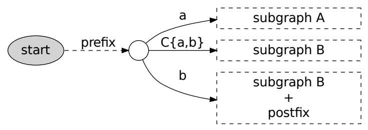


(a) Test Trace to be Added (b) New Trace Graph

Figure 4. Add a Trace with a Matching Default

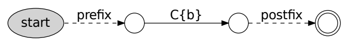


(a) Test Trace to be Added (b) Modify the Trace Graph: Split the Default Branch

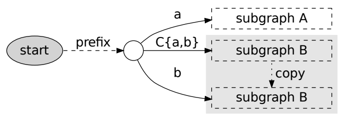


(c) New Trace Graph

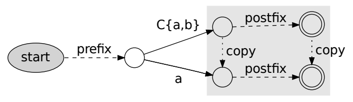
Figure 5. Add a Trace with a Subset of the Default



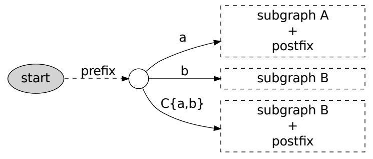
(a) Test Trace to be Added



(b) Modify the Trace Graph: Split the Default Branch



(c) Modify the Test Trace: Split the Default Branch



(d) New Trace Graph

Figure 6. Add a Trace with a Differing Default

the postfix of the trace to the subgraph of the default already in the trace graph. Figure 4 illustrates this. Figure 4a shows the test trace to be added. The prefix of the trace matches the prefix of the trace graph (see Figure 3) and the complementary set $\complement\{a\}$ matches the complementary set in the trace graph. Therefore, the postfix of the trace has to be added to the subgraph of the complementary set. Assuming that there are no other defaults in the postfix, this is done according to the construction rules described in Section IV-B. Figure 4b depicts the resulting trace graph after the new trace was added.

In the second case, the new trace matches a *subset* of the complementary set in the trace graph. The situation is depicted in Figure 5, the signal following the prefix in the trace (Figure 5a), *b*, is a subset of the complementary set $\complement\{a\}$. However, the postfix cannot simply be added to the subgraph of the complementary set, as this would allow unspecified traces. Instead, before adding the postfix, the trace graph is modified as shown in Figure 5b. The signal *b* is removed from the complementary set and represented by a distinct edge. Now, the new trace matches exactly and the adding proceeds as described for the first case. Figure 5c shows the result.

In the third and last case, the complementary sets of the new trace and the trace graph *overlap* (see Figure 6). The trace contains an edge marked with the complementary set $\complement\{b\}$ (Figure 6a), whereas the trace graph contains an edge marked with the complementary set $\complement\{b\}$ (see Figure 3). The complementary set of the test trace to be added does not fit the complementary set of the trace graph, but there is an overlap, i.e., every signal which is neither *a* nor *b* matches both sets.

The solution is similar to the second case. The transitions in the trace need to match the transitions in the trace graph, so the sets are split accordingly. For the trace graph, the edge marked *b* is branched out from the complementary set (Figure 6b). The remaining complementary set in the trace graph is $\complement\{a, b\}$. However, the complementary set of the test trace still does not match, so the test trace is also split (Figure 6c). The complementary sets of the trace graph and the test trace are now identical, $\complement\{a, b\}$, but the test trace has been split into two test traces. Now, the two resulting test traces can be added to the trace graph, resulting in the trace graph shown in Figure 6d.

The described techniques also generalize to sets with more than one element. In this case, the sets associated with the split branches are determined as the intersections and differences of the given sets.

VI. Experimental Results

To assess the power of our learning approach, we have developed a prototypical implementation [11]. The implementation realizes an Angluin-style learner, which is adapted to learning from test cases, and the organization of the test data into a trace graph as discussed in Sections IV and V. Using the prototype, we performed a case study based on the *conference protocol* [12]. The conference protocol describes a chat-box program that can exchange messages with several other chat-boxes over a network.

Table I shows our results for a simple version of the conference protocol, where the sequence of the signals was fixed. The protocol scaled according to the number of participating chat-boxes. As the table shows, the semantic state-merging reduces the size of the trace graph by more

Number of	Size of	Size of the Trace Graph		Size of the Test Suite	
Chat-Boxes	Target Automaton	Without Merging	With Merging	Without Merging	With Merging
1	72 edges, 8 nodes	33 nodes	13 nodes	6 **pass** traces	2 **pass** traces
2	168 edges, 12 nodes	60 nodes	22 nodes	9 **pass** traces	3 **pass** traces
3	304 edges, 16 nodes	90 nodes	30 nodes	12 **pass** traces	4 **pass** traces
4	480 edges, 20 nodes	120 nodes	40 nodes	15 **pass** traces	5 **pass** traces
5	696 edges, 24 nodes	164 nodes	48 nodes	18 **pass** traces	6 **pass** traces

Table I
EFFECT OF SEMANTIC STATE-MERGING

than half in this example, while the learned automaton was identical. Also, the test suite can be smaller. In addition, the compact version of the trace graph also allows an optimized equivalence query.

Additional experiments show that while our approach quickly learns clearly structured models, models with a high degree of variation are hard to learn and require a large test suite. In fact, for a more complex version of the conference protocol with variable signal sequence, the model could only be reconstructed from a test suite satisfying path coverage [11]. However, it is possible that the size of the test suite can be further reduced using additional state-merging rules, e.g., marked stable testing states.

VII. CONCLUSION

We have presented a learning approach that combines state-merging and learning techniques to generate a DFA from a test suite. The state-merging is used to represent the test suite and to find additional test cases exploiting the semantic properties of the test language. The combined approach has been implemented in a prototypical tool. Experiments show that while the state-merging approach reduces the size of the test suite needed for correct identification of the model, complex models still need a large number of test cases for correct identification.

Optimizations to deal with this problem comprise the extension of the semantic state-merging approach to exploit further information contained in the test cases and an extension of the learning algorithm to work with unanswerable membership queries. In addition, the relation between test suite coverage, system structure, and learnability offers interesting research topics. Based on the experiments with our learning approach, the next step is to incorporate the identified optimizations into our prototypical implementation. In the long run, our findings on the learnability of different models could also be used to assess the adequacy of a test suite.

REFERENCES

[1] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[2] E. Werner, S. Polonski, and J. Grabowski, "Using Learning Techniques to Generate System Models for Online Testing," in *Proc. INFORMATIK 2008*, ser. LNI, vol. 133. Köllen Verlag, 2008, pp. 183–186.

[3] M. Shahbaz and R. Groz, "Inferring Mealy Machines," in *Proc. FM 2009*, ser. LNCS, vol. 5850. Springer, 2009, pp. 207–222.

[4] F. Aarts, B. Jonsson, and J. Uijen, "Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction," in *Proc. ICTSS'10*, ser. LNCS, vol. 6435. Springer, 2010, pp. 188–204.

[5] T. Berg, B. Jonsson, and H. Raffelt, "Regular Inference for State Machines Using Domains with Equality Tests," in *Proc. FASE 2008*, ser. LNCS, vol. 4961. Springer, 2008, pp. 317–331.

[6] T. Bohlin, B. Jonsson, and S. Soleimanifard, "Inferring Compact Models of Communication Protocol Entities," in *proc. ISoLA 2010*, ser. LNCS, vol. 6415. Springer, 2010, pp. 658–672.

[7] M. Shahbaz, K. Li, and R. Groz, "Learning and Integration of Parameterized Components Through Testing," in *Proc. TestCom 2007*, ser. LNCS, vol. 4581. Springer, 2007, pp. 319–334.

[8] J. Esparza, M. Leucker, and M. Schlund, "Learning Workflow Petri Nets," in *Proc. PETRI NETS 2010*, ser. LNCS, vol. 6128. Springer, 2010, pp. 206–225.

[9] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker, "SMA — The Smyle Modeling Approach," *Computing and Informatics*, vol. 29, no. 1, pp. 45–72, 2010.

[10] *ETSI Standard (ES) 201 873: The Testing and Test Control Notation version 3; Parts 1–10*, ETSI Std., Rev. 4.2.1, 2010.

[11] E. Werner, "Learning Finite State Machine Specifications from Test Cases," Ph.D. dissertation, Georg-August-Universität Göttingen, Göttingen, Jun. 2010. [Online]. Available: http://webdoc.sub.gwdg.de/diss/2010/werner/

[12] L. D. Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. F. E. Belinfante, and R. G. Vries, "Formal Test Automation: The Conference protocol with TGV/Torx," in *Proc. TestCom 2000*, ser. IFIP Conference Proceedings. Kluwer Academic Publishers, 2000, pp. 221–228.

Generic Data Format Approach for Generation of Security Test Data

Christian Schanes, Florian Fankhauser, Stefan Taber, Thomas Grechenig
Vienna University of Technology
Industrial Software (INSO)
1040 Vienna, Austria
E-mail: christian.schanes,florian.fankhauser,
stefan.taber,thomas.grechenig@inso.tuwien.ac.at

***Abstract*—Security testing is an important and at the same time also expensive task for developing robust and secure systems. Test automation can reduce costs of security tests and increase test coverage and, therefore, increase the number of detected security issues during development. A common data format as the basis for specific test cases ensures that the implementation of the generation logic for security test data is only needed once and can be used for various data formats by transforming the data to the common data format, generating the test data and transforming back to the original data format. The introduced approach enables to generate test data for various formats using a single implementation of the generation algorithm and applying the results for specific test cases in different data formats.**

Keywords—Software testing; Computer network security; Fuzzing.

1. Introduction

Software systems are getting more and more complex today using various programming languages and protocols developed by many different companies. Security testing of such systems is required to increase robustness against attacks. The extent of the attack surface of systems with different interfaces requires multiple different tools for security testing, which also increases required resources for test execution. The generation of the required test data and the execution of security tests are time consuming and, therefore, expensive. Automatization of test execution is required to reduce costs and increasing the test coverage of a System Under Test (SUT).

Applications for conducting security tests require test data as input, which can be mutated using methods to alter the data to structures and values that are specific for security tests. For generating such information a possible available input format definition can be used, e.g., XML Schema Definition (XSD). Our approach considers the automated extraction of such a format definition based on a set of test data if such a definition is not available or not accurate enough for generating proper test data. Such test data can be gained during development or by using input data of functional tests.

This work presents an approach using Extensible Markup Language (XML) as generic data format for test execution. Using one generic format allows the implementation of one smart generation algorithm instead of using specific algorithms for various data formats and allows to use a reduced tool set for test execution. For additional protocols only new transformation rules are required and not the new implementation of a completely new generation algorithm.

For testing a single service multiple tools are required to test the various layers of the service. For example, a web service uses Hyper Text Transfer Protocol (HTTP) as transport protocol, XML or JavaScript Object Notation (JSON) for data transport and in security critical environments cryptographic methods are used to ensure protection goals, which often require X.509 certificates. Our approach allows to use one tool to generate test data for all of the used protocols in a web service. As description language we use XSD [1] introduced by W3C as a standard for describing how a certain XML document is supposed to look like. As a prototype we implemented transformation routines for Abstract Syntax Notation One (ASN.1), which is used for many Public Key Infrastructure (PKI) protocols of the X.509 standard like revocation lists or certificates [2].

The remainder of this paper is structured as follows: Section 2 lists related work. Section 3 discusses XML as a common format for security tests. Section 4 gives details about the introduced approach for generating test data. The implemented prototype was used for generating test data for PKI protocols, which is presented in Section 5. The paper finishes with a conclusion and ideas for further work in Section 6.

2. Related Work

Thompson [3] defines security failures as side effects of the software, which are not specified and make security testing hard. Fuzzing is one possible technique to find such side effects by executing the application with many automatically randomly or rule-based generated input data [4]–[6].

If no detailed specification of the interfaces are available to generate data automatically it is possible to extract test data from executed functional tests, e.g., by using stored test data, network traffic [7], [8] or by dynamic binary analysis [9]–[11].

Transforming from different data formats to XML and vice versa is discussed by various authors, e.g., for Resource Description Framework (RDF) [12], relational databases [13], [14] or ASN.1 [15]. ASN.1 [16] is a format commonly used by different protocols and applications. Yoon et al. [17] discuss the usage of ASN.1 for Simple Network Management Protocol (SNMP). The ASN.1 format is also used for various PKI protocols like X.509 [2].

Moreover, different authors discussed the similarity of ASN.1 and XML [18]–[22] and mapping of ASN.1 and XML [21].

Due to the widespread usage of XML processing systems the generation of test data was discussed by various authors. Different approaches are available, which are based on different sources for generation like document specification languages (e.g., XSD or Document Type Definition (DTD)), based on specific generation rules in a non XML format or by using example input data. Aboulnaga et al. [23] discuss a generator, which is based on simple implemented rules, which allow limited generation of XML data. Bertolino et al. [24]–[26] implemented TAXI, a XML generator, which generates documents (instances) based on a given XML schema and follows the XML-based Partition Testing approach (XPT), which is a modification of the Category Partition Method. Xu et al. [27] also use a XML schema as basis to generate valid and invalid messages for testing web services. ToXgene by Barbosa et al. [28] is a template based XML generator, which uses a template specification language within a XML schema to describe the rules for generating data. Pan et al. [29] use example instances to extract allowed values for the generated data additionally to the document specification.

3. XML as Common Format for Test Data Generation

Implementing techniques for optimized data generation for tests is often required for test execution to detect as many errors as possible with as less effort as possible. For this our approach considers the usage of a common data format where optimized algorithms for data generation have to be implemented only once.

3.1. Definition of Security Test Data

The attack surface of a SUT contains various interfaces using different protocols and data formats on various layers, e.g., network or application. Analyzing the attack surface of an application is important for conducting security tests. The application takes the input data and, often, at first performs syntactical validation of the data. For state based protocols another validation is done by the state machine, which only allows specific state transitions. Finally, the data will be handed over to the business logic, which processes the values. Understanding and considering such validation steps is important to test the intended piece of software within the application, which is only seen as black box.

For preparing input data for security tests, semantical, syntactical and state aspects have to be considered. Semantical aspects consider a standard compliant processing of the information. The generated values have to fulfill certain restrictions, e.g., a valid checksum. The definition can be stated in the technical interface specification, e.g., restrictions in XSD, or restrictions in the applications without explicit definition in the interface specification. Both aspects have to be considered. For testing semantical aspects structural and state valid data is required because otherwise the data will be discarded during the validation process of the SUT and the business logic, which semantically processes the data will not be triggered. State aspects consider state based protocols where the underlying state machine expects data in a specific order and otherwise discards it. The structure is considered by validating the syntax. It is the most basic aspect of the SUT interface because a structural invalid message will be rejected early before handling the information to the state machine or the business logic. It is required to generate structural valid documents according to the interface specification and structural invalid documents to test the robustness of the system with such data. For increased coverage of the SUT all of the mentioned aspects have to be considered when creating test data for security tests.

Common practice for tests is to inject only one fault per test case to ensure reproducibility [30]. This applies for semantical, syntactical as well as for state aspects. Therefore, valid data is required for security tests to be able to inject faults on specific positions only.

3.2. Data Generation Approach Based on XML

Figure 1 shows the test data generation process used in our approach. There are several alternative paths depending on the available source information for producing test data. One path is based on existing data definition languages, which are used directly if they are already available as XSD. If they are available in a different format they can be transformed to the used XSD format. Another path is the usage of available input data, which will first be transformed to XML and based on XML input data a data definition will be derived as XSD. The generation is always based on the given or generated XSD and an optional set of rules or example data to produce accurate values for the test data.

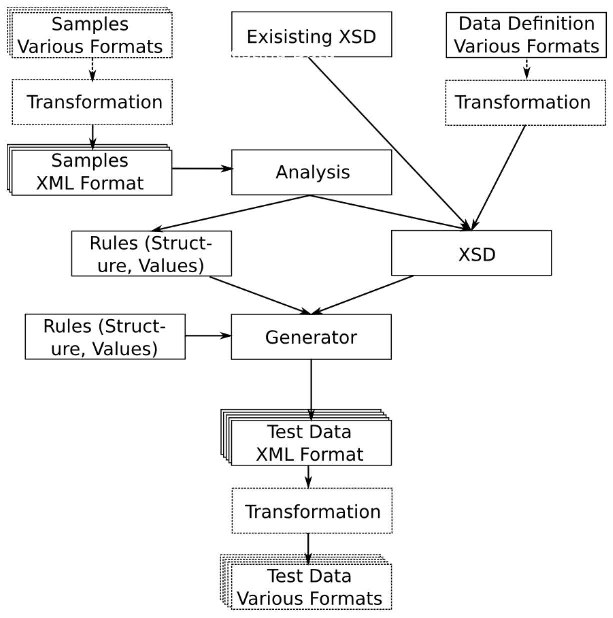


Figure 1. Process of Test Data Generation Approach Based on a Common Data Format

3.3. Using Transformation to Generate Data for Various Formats

Transformation is one of the key aspects of our approach. By transforming from various data formats to XML/XSD a generic data generation approach is possible. For many different data formats there already exist methods for transforming to and from XML [12]–[14], [20], [21], [31].

The approach also allows transformation of data definition languages, e.g., DTD. This allows the generation of test data if a data definition is available, which can be transformed to XSD. After the generation process all the produced XML test data will be transformed to the destination data format.

3.4. Extraction of Interface Definition Based on Samples

Additionally, the generation is also possible without a document specification as can be seen in Figure 1, e.g., based on a set of input data extracted from the network.

A way to find or rather deduce some kind of schema is to use a set of XML input data samples. Only valid XML instances can be used as samples. Such data can be identified, for example, by executing the application and logging the generated data.

The approach uses the given samples to deduce rules which the XML (its structure and content) follow and build a specification as XML schema. The resulting XML schema can then be used for the test data generation.

The samples usually can not cover every possible data variation that is actually allowed and some assumptions have to be made when deducing rules for a common input specification. This means that some information about the actual input specification gets lost and, therefore, the data that can be generated will be restricted.

4. Automated Generation of Test Data

The presented approach for generating test data uses XSD as source, which builds the model to generate the test data. Based on the given XSD, structural valid and invalid XML documents with valid and invalid values can be generated. XML is hierarchically structured, which allows to systematically traverse through the hierarchical tree, starting at the root-element. Each node (e.g., sequences, elements, attributes, ...) of the tree will be processed and based on schema details, e.g., `min-`/`maxOccurs`, choices, various possible facets, etc. the test data will be generated.

The generated test data will be used for security testing by using a fuzzing approach. Two different aspects are considered during test data generation to allow usage by the fuzzing engine. Firstly, valid test data, which can be further used to inject faults are generated. This requires valid structures with valid values to avoid discarding by validation routines in the SUT. Secondly, a set of test data is required that is using invalid structures, which do not fulfill the available data definition.

4.1. Approach for Generation of Valid XML Structures

In XML schema it is possible to define customized data types in form of simple and complex types. These types can then be used like built-in data types to specify the content of elements or attributes. Simple types are used to constrain existing data types. They do not allow the definition of attributes or child elements. Complex types are used for defining a structure of elements with possible further child elements, which can again be simple or complex types. There are three indicators for ordering or choosing these elements, which are `sequence`, `all` and `choice`. The `sequence` indicator specifies that the elements have to be in a specific order, which has only one valid variant for ordering the elements, because every other sequence of the given elements makes the document schema invalid.

The order of the defined elements is irrelevant if the `all` indicator is used. Every permutation of those elements within the `all` indicator would represent a valid variant of the complex type.

The `choice` indicator differs from the others in the way that from the child elements only one element can be chosen. Therefore, there is one valid data variation for each element defined in the `choice`.

XSD further supports to define elements and attributes `optional` or `required`. If an element is marked as optional, then there exist at least two valid variants for this element. One with the element and one without it. In case of elements, it is also possible to define the occurrence of such an element by using the facets (restrictions or "rules" defined in XSD) `minOccurs` and/or `maxOccurs`, which also has to be considered for the generation of data. If such restrictions exist, a variant with the minimum occurrence, one with the maximum occurrence and one with an occurrence between those two limits are chosen. If the restriction of `maxOccurs` is "unbounded" a predefined maximum value has to be defined for the element.

The result of the test data generation in form of the generated XML instances in the end is basically just the combination of the available valid variants of each element to form a number of different XML documents. These XML documents will all be valid according to the given XML schema, because they were produced by applying every rule defined in the schema and using them to find meaningful variants based on the existing restrictions.

4.2. Approach for Generation of Schema Invalid Variants

In addition to the generation of valid XML documents, in this section different approaches for manipulating the structure of an XML document are explained, so that it is not valid according to its XML schema. Such instances are required for testing SUT behavior during processing schema invalid data. For the presented approach only invalid variants are possible, which are describable in XSD and no manipulated instances are generated, e.g., non XML well formed documents, because such documents can not be transformed back to the required data format, e.g., ASN.1.

For generating invalid variants only complex types and indicators are considered. Generating invalid simple types are not required in the used approach because this is done by the fuzzing approach.

The order of the elements within a `sequence` is crucial, which means that the elements can be swapped to produce an invalid XML instance. In the case of an `all` or a `choice` the order of the elements is irrelevant so it is not possible to produce invalid instances when shuffling the order. Invalid instances for all indicators are possible by inserting not allowed elements.

The `maxOccurs` and `minOccurs` attributes are used for producing an invalid XML instance with a wrong number of elements. For that, equivalence partitioning is applied. Moreover, specific security critical values are used as, for example, byte ranges. The same approach applies to `maxOccurs` and `minOccurs` within an indicator as well. For example, if a choice is used in which `minOccurs` and `maxOccurs` are 1, no item or several items are selected from the choice so that the XML instance is invalid.

Attributes can be optional or mandatory by using the attribute `fixed` in XSD. For mandatory attributes an invalid instance is given by skipping the attribute.

4.3. Generation of Proper Values

The generation of proper values is important to get valid XML documents for test execution. The documents will further be used as input in a fuzzing engine, which uses fault injection to prepare the final security test data.

For the generation the definitions of the XSD are used, which provide information about data types and specific constraints. Additionally, it is possible to provide a set of samples to extract proper values from.

The XML schema defines built-in primitive and derived types. For many types it is possible to define further constraining facets for their value space in the XML schema definition. For example, `string` allows the facets listed in Figure 2. During generation special aspects (e.g., format, value space and facet/restrictions) of all types have to be considered.

Facet	Description
length	For a fixed length string
minLength	Minimum length of the expected string
maxLength	Maximum length of the expected string
pattern	Regular expression restriction
enumeration	Only allow a set of possible values
whiteSpace	Definition of handling white spaces

Figure 2. Relevant Facets for a String Type in XSD

If samples are given for the generation process as shown in Figure 1 then the values from the samples are extracted. The example values can be further used for building valid test data. For this process the XML samples are parsed and each value is stored in a list together with the position of the value as XPath statement. Additionally, it is possible to use predefined values from a configuration file again as a tuple of the position as XPath statement and the value. This approach allows manual overriding of the automatic value generation.

For specific data it is possible to use a predefined generator implementation, which will build valid data, e.g., timestamps, unique IDs, ... Additionally, the used fuzzing engine supports the extraction of values of the communication protocol, which is required for further input data, e.g., session ID in the response from the server.

5. Prototype Implementation: Generating Data for PKI

We implemented the presented approach as a prototype and used it for generating data for ASN.1 based PKI protocols. Our approach uses the BouncyCastle library to read the ASN.1 structure. We implemented the transformation between ASN.1 and XML and applied the approach for generating data for Online Certificate Status Protocol (OCSP) messages and X.509 certificates.

5.1. ASN.1 and XML

ASN.1 supports a number of different encoding rules, e.g., Basic Encoding Rules (BER) or Distinguished Encoding Rules (DER). The considered encodings are Type Length Value (TLV) based structures where type is a unique identifier for the type and basically resembles the element name in XML documents (e.g., `<TBSCertificate>`). Instead of using closing elements as used by XML ASN.1 defines the length of the value. The value can be a simple or complex type, which is again another TLV structure, which is similar to those of XML documents.

5.2. Fault Injection for Generating X.509 Certificates

X.509 certificates are used for many scenarios for securing infrastructures, e.g., Virtual Private Network (VPN) or Secure Socket Layer (SSL). Thorough testing of such security gateway implementations is important to ensure secure and robust implemented security functionality. We generated X.509 test certificates for security testing infrastructures. Therefore, we used samples available from functional tests to generate security test data. Figure 3 shows the process of generating certificates using a fuzzing approach for fault injection.

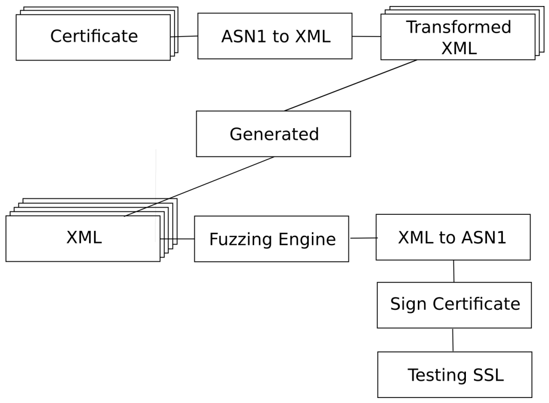


Figure 3. Process of Certificate Fuzzing Using XML

For testing structural robustness problems schema invalid variants were generated as discussed in Section 4.2. Faults for value fields for simple data will be injected by the used fuzzing engine based on random generated data and predefined attack vectors. This is done for each field in the X.509 certificate. The used transformation implementation additionally allows fuzzing of the ASN.1 length and type fields to ensure robustness of internal ASN.1 parsing functionality by manipulating the values so that the defined type and length are not fitting the content of the field. The algorithm for generation of the data considers the certificate content, e.g., key, subject, and the signature part, e.g., signature value, signature algorithm.

Finally, for X.509 certificates a valid signature is required. For this a valid signature will be attached to the certificate after transforming back to ASN.1. This allows the usage of the certificate for the tested use case, e.g., in Figure 3 the certificate will be used to establish a SSL connection.

6. Conclusion and Further Work

We presented an approach for using XML as a common format for the generation of security test data, which allows to test semantical, syntactical and state aspects. For the generation of security test data various data formats can be transformed to and from XML, which allows testing different protocols by implementing the generation algorithm for the security test data only once. This reduces costs and allows the introduction of a framework for the generation of test data. For new formats only the transformation routines have to be implemented.

The approach was applied for security testing ASN.1 based PKI protocols. As samples for the generation algorithm X.509 certificates of the functional tests were used. The process started by transforming ASN.1 based X.509 certificates to XML, generating test data and transforming back to ASN.1 for testing a SSL implementation.

For future work the approach should be extended by automatically detecting semantical constraints, which are not defined in the XSD to produce semantical valid test data. Currently, the performance of the implemented prototype leads to long running generation tasks. A more efficient implementation is required to increase the number of data variations for the test execution.

References

[1] W3C, “Xml schema,” http://www.w3.org/TR/xmlschema-0/, [accessed: 2010-10-20].

[2] Y. Turcotte, O. Tal, S. Knight, and T. Dean, “Security vulnerabilities assessment of the x.509 protocol by syntax-based testing,” vol. 3, Oct./Nov. 2004, pp. 1572–1578.

[3] H. H. Thompson, “Why security testing is hard,” *IEEE Security & Privacy Magazine*, vol. 1, no. 4, pp. 83–86, 2003.

[4] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[5] J. E. Forrester and B. P. Miller, “An empirical study of the robustness of windows nt applications using random testing,” in *WSS’00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 6–6.

[6] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing,” in *PLDI ’08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2008, pp. 206–215.

[7] O. Udrea, C. Lumezanu, and J. Foster, “Rule-based static analysis of network protocol implementations,” *Inf. Comput.*, vol. 206, no. 2-4, pp. 130–157, 2008.

[8] W. Cui, J. Kannan, and W. Helen, “Discoverer: automatic protocol reverse engineering from network traces,” in *SS’07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–14.

[9] J. Caballero, H. Yin, Z. Liang, and D. Song, “Polyglot: automatic extraction of protocol message format using dynamic binary analysis,” in *CCS ’07: Proceedings of the 14th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2007, pp. 317–329.

[10] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Briz, “Tupni: automatic reverse engineering of input formats,” in *CCS ’08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 391–402.

[11] Z. Lin and X. Zhang, “Deriving input syntactic structure from execution,” in *SIGSOFT ’08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2008, pp. 83–93.

[12] D. Van Deursen, C. Poppe, G. Martens, E. Mannens, and R. Walle, “Xml to rdf conversion: A generic approach,” nov. 2008, pp. 138 –144.

[13] J. Fong, F. Pang, and C. Bloor, “Converting relational database into xml document,” 2001, pp. 61 –65.

[14] M. Jacinto, G. Librelotto, J. Ramalho, and P. Henriques, “Bidirectional conversion between xml documents and relational databases,” 2002, pp. 437 – 443.

[15] ITU-T, “X.693 information technology asn.1 encoding rules: Xml encoding rules (xer),” X SERIES: DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY OSI networking and system aspects - Abstract Syntax Notation One (ASN.1), Nov. 2008, identical standard: ISO/IEC 8825-4:2008 (Common).

[16] ——, “X.680, Abstract syntax notation one (ASN. 1): Specification of basic notation,” 1994.

[17] J. Yoon, H.-T. Ju, and J. Hong, “Development of snmp-xml translator and gateway for xml-based integrated network management,” *Int. J. Netw. Manag.*, vol. 13, no. 4, pp. 259–276, 2003.

[18] D. Mundy and D. Chadwick, “An xml alternative for performance and security: Asn.1,” *IT Professional*, vol. 6, no. 1, pp. 30–36, 2004.

[19] D. Mundy, D. Chadwick, and A. Smith, “Comparing the performance of abstract syntax notation one (asn.1) vs extensible markup language (xml),” in *In Proceedings of the Terena Networking Conference*, 2003.

[20] T. Imamura and H. Maruyama, “Mapping between asn.1 and xml,” *Applications and the Internet, IEEE/IPSJ International Symposium on*, vol. 0, 2001.

[21] A. Triglia, “The asn.1 language as a new schema definition language for xml,” XML Europe 2002 Conference, May 2002.

[22] ITU-T, “X.690: ITU-T Recommendation X.690 (1997) Information technology-ASN.1 encoding rules: Specification of Basic Encoding Rules (BER),” 1997.

[23] A. Aboulnaga, J. F. Naughton, and C. Zhang, “Generating synthetic complex-structured xml data,” in *In Proc. 4th Int. Workshop on the Web and Databases (WebDB2001*, 2001.

[24] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, “Taxi– a tool for xml-based testing,” in *ICSE COMPANION ’07: Companion to the proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 53–54.

[25] ——, “Systematic generation of xml instances to test complex software applications,” in *RISE’06: Proceedings of the 3rd international conference on Rapid integration of software engineering techniques*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 114–129.

[26] ——, “Automatic test data generation for xml schema-based partition testing,” in *Automation of Software Test , 2007. AST ’07. Second International Workshop on*, may. 2007, p. 4.

[27] W. Xu, J. Offutt, and J. Luo, “Testing web services by xml perturbation,” in *ISSRE ’05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 257–266.

[28] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons, “Toxgene: a template-based data generator for xml,” in *SIGMOD ’02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2002, pp. 616–616.

[29] C.-C. Pan, K.-H. Yang, and T.-L. Lee, “A flexible generator for synthetic xml documents,” in *Proceedings of the International Conference on Information Networking (ICOIN 2003)*, 2003, pp. 1232–1239.

[30] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[31] E. Day, “The use of asn.1 encoding rules for binary xml,” http://www.obj-sys.com/docs/ASN1forBinXML.pdf (last accessed: August 15, 2011).

A Classification for Model-Based Security Testing

Michael Felderer, Berthold Agreiter, Philipp Zech and Ruth Breu
Institute for Computer Science
University of Innsbruck
Technikerstr. 21a, A-6020 Innsbruck, Austria
{michael.felderer, berthold.agreiter, philipp.zech, ruth.breu}@uibk.ac.at

***Abstract*—Security testing defines tests for security requirements of software. Security requirements are non-functional, and thus require a different way of testing compared to functional requirements. Model-based testing applies model-based design for modeling test artifacts or the automation of test activities. Although model-based testing techniques improve security testing, these two testing activities have rarely been combined systematically. Like functional system models improve functional testing, risk models can improve security testing. This paper first gives an overview of existing security testing approaches, and based on that, develops a novel classification for model-based security tests along the two dimensions risk and automated test generation. The classification allows for understanding which areas of model-based security testing are already well-covered by research and practice, and furthermore, can serve as a guideline for deciding which testing approach fits specific circumstances. Based on the classification, we identify tasks for interesting future research.**

***Keywords*-Secure Systems, Verification and Testing, Security Testing, Model-based Testing**

I. Introduction

Security testing means to test for security requirements of software. By the increasing use of software in more and more areas of our daily life, also the amount of sensitive data which is processed automatically, e.g., in electronic health or e-government applications, increases. Consequently, the adherence to security requirements, which aim to protect sensitive data and the systems processing that data, is constantly gaining importance. Several classifications of security requirements can be found in literature, e.g., [1]–[3]. The most prominent list of security requirements (cf. [3]) distinguishes six types of security requirements:

- *Confidentiality* is the assurance that information is not disclosed to unauthorized individuals, processes, or devices.
- *Integrity* is provided when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed.
- *Authentication* is a security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's authorization to receive specific categories of information.
- *Authorization* provides access privileges granted to a user, program, or process.
- *Availability* guarantees timely, reliable access to data and information services for authorized users.
- *Non-repudiation* is the assurance that none of the partners taking part in a transaction can later deny of having participated.

Security testing aims at checking whether these requirements are satisfied under various conditions. Due to the openness of modern service-oriented systems, security testing has gained much interest in the last years [4] and has become a vast field of research.

Model-based testing (MBT) applies model-based design for the *modeling of test artefacts*, or the *automation of test activities* [5]. MBT supports the early definition and automatic validation of tests on the abstract model level. Most of today's model-based testing approaches consider the automated generation of test cases from a functional system description. MBT itself is well-covered in literature, and many tools are already on the market applying model-based approaches [6]. Furthermore, also partial test models are often encountered in practice, where domain expertise by a test engineer is needed for designing tests. Although model-based testing increases the level of abstraction in different aspects, supports a systematic, model-based, a-priori security test design, and reduces the required expertise for security testing, it is not widely used for testing security requirements today. So far, *model-based security testing* (MBST), i.e., model-based testing of security requirements, is more or less only used for testing access control policies in academia (see Section II on security testing approaches).

MBT re-uses functional system knowledge which is provided by models, so that the test engineer can abstract from many aspects in that respect. However, for testing security requirements, the test engineer further needs security knowledge for being reasonably able to design tests. Since security is tightly coupled with risk, risk models can potentially fill this gap. Based on an overview of existing classical and model-based security testing approaches, the contribution at hand defines a novel classification of model-based security testing approaches, and systematically identifies interesting future research directions to promote model-based security testing. We classify model-based security testing approaches along the two dimensions *risk* and *automated test generation*. For each category of security tests, we describe typical approaches and identify fields which offer the potential to put the discipline of security testing forward.

Since we want to classify security testing approaches in this contribution, we first detail what *security* is. Security is a *non-functional* property, and defines how a system is supposed to be, in contrast to what a system is supposed to do. Security threats are caused by faults and flaws. *Faults* may lead to *failures* which harm security requirements, and *flaws* are security problems which may lead to *vulnerabilities*. Security can be classified into three main levels. Each level exhibits its own security threats, but also offers the corresponding security requirement to deal with these threats. The three security levels network security, operating system security, and application security (cf. [7]) are characterized as follows:

- *Network Security* involves tackling threats which target the network. The main threats are (distributed) denial-of-service, network intrusion, or attacks during message transport (cf. [8]).
- *Operating System Security* is related to the basic services of operating systems and includes protection against all sorts of malware (virus, worm, spyware, etc.). The protection mechanisms involve antivirus, anti-malware, operating system level access control mechanisms, firewalls, etc. (cf. [9])
- *Application Security* deals with the threats targeting a specific application. It includes unauthorized access, information theft, and misuse of the application. The security mechanisms include access control mechanisms and policies, application level encryption/decryption, etc.

All these security levels can be subject to software tests. In this paper, we focus on application security and disregard the network and operating system, nevertheless this does not limit the applicability of the classification we introduce later.

The remainder of this paper is organized as follows: In Section II, we give an overview of existing security testing approaches. In Section III, we define a novel classification of model-based security testing with the dimensions risk and automated test generation. Finally, in Section IV, we summarize and sketch future work in the area of model-based security testing arising from our classification.

II. Security Testing Approaches

In this section, we give a representative overview of actual security testing approaches to motivate model-based security testing that considers risks and partial test generation. Security testing is often fundamentally different from traditional software testing because it emphasizes what an application should not do rather than what it should do. This fact was also pointed out by Alexander in [10], where the author distinguishes between positive and negative requirements modeled as use cases and misuse cases.

For testing positive security requirements, i.e., functional security properties that are defined in the requirements specification, classical functional testing techniques can be applied (Michael and Radosevich [11] provide a detailed listing of functional testing techniques for testing positive security requirements, e.g., equivalence testing or decision tables). Testing positive security requirements can be performed by the traditional test organization [12].

Negative security requirements express what a system should not do, respectively what should not happen. The set of negative requirements is therefore infinite on principle, and this makes it impossible to achieve complete test coverage. A promising way to overcome this problem is the derivation of tests based on a risk-analysis [11]. Due to this fact, risk-based testing (RBT) techniques [13] are highly relevant for security testing [14]. Based on a threat model, or based on abuse cases [15], vulnerabilities can be identified and prioritized relying on a risk analysis.

Tests can be designed in a classical operational way or, very frequently, as penetration tests which attempt to compromise the security of a system [16] by acting like an attacker trying to penetrate the system and exploit its vulnerabilities. Specifically, it tests missing functionality or side-effects of the system. Often also the environment of a system is the target of attacks instead of the system itself (e.g., exploiting an unpatched operating system). Furthermore, there are several penetration testing strategies [17], e.g., internal, external, or blind testing strategy, and penetration testing tools available [18], e.g., port or vulnerability scanners. Moreover, several standards for penetration testing exist, among which the Open Source Security Testing Methodology Manual (OSSTMM) [19] is the most prominent one. The OSSTMM methodology covers the whole process of risk assessment involved in a penetration test, from initial requirements analysis to report generation.

Besides penetration testing, another well-known approach to functional security testing is fuzzing [20]. The very basic idea behind a fuzzer is to test a protocol implementation on possible security flaws due to improper handling of malicious input. However, as Takanen et al. show in their book [20], fuzzing may also be used for testing other types of software in terms of security. Yet, what makes fuzzers difficult to use is the fact that a fuzzer by design cannot be general-purpose. Hence, for each new software to test, a specific fuzzer has to be implemented from scratch, which in fact is a challenging task. For instance, Taber et al. [21] present a fuzzer dedicated to security testing SIP-based VOIP applications. Yet, fuzzers still suffer from their randomness, put another way, testing with a fuzzer lacks all aspects of a structural approach. However, as done in [22], combining the idea of fuzzing with the concept of model–based testing, allows for systematic and automated testing of software applications.

On a more abstract level, model-based testing approaches have been applied for testing access control policies.

In [23], the authors describe a model–based testing approach for checking whether access control policies are properly enforced by the system under test (SUT). The functional model is written in the B language [24] and used for the security test generation from so called *test purposes*. Test purposes are defined as regular expressions and describe a general sequence of operation calls to induce a certain situation on the SUT. The approach aims at the automated generation of test cases from the SUT.

A similar approach is used in [25] for testing smart cards where the access control rules are defined by test purposes. Another model-based approach for testing role-based access control policies is presented in [26]. The creation of test targets (actual access requests) is based on three different strategies: (1) only taking into account roles and permissions, (2) considering all rules in a policy and (3) completely at random. The tests target the policy decision point to check whether its decisions are correct or not.

Usually, models that are used for test generation are supposed to be correct and attack-resistant. In an attack-driven approach, a common practice is to modify the model such that it contains errors that can be revealed by attacks. The model mutation makes it possible to simulate attack scenarios on the model such that the response provides an observable answer to the attack. If a test representing an attack is executed against the SUT and reacts as expected from the modified model, then a vulnerability has been identified. In the mutation-based testing approach of DeMillo et al. [27], modified models are used in the standard test generation process. The mutation guides the test generation with a focus on the introduced errors.

Traon et al. [28] present a set of security mutations in access control policies used to drive the test generation. The generated tests are used to request access to secure data. The success of the access, allows to conclude the presence of failures in the SUT.

Jurjens [29] uses fault-injection techniques to introduce security faults in UMLsec models [30], and model-checkers are then used to build traces leading to the faults. The traces are then used as test cases for the system.

The model-based testing approaches mentioned above focus on the test generation from *complete* models where all security tests are derived fully automated from a formal model. Such models are costly to create and hard to maintain, and therefore only rarely applied in practice. Thus, we also consider *partial* security test generation where some security tests are generated automatically from a security model, and others are added manually. In classical approaches no test models are created, and the automatic test generation is *missing*.

Fig. 1 shows and compares the three degrees of automatic security test generation from software (components), i.e., complete, partial, and missing.

In the case of complete test generation, all tests are generated from a formal security model which is depicted as graph in Fig. 1. In the case of partial test generation, tests are generated from a formal security model and manually which is depicted by a graph and a cloud as source of test generation. The graph and the cloud are linked to show that the design of the test model and the manual test definition informally influence each other. Finally, if automatic test generation is missing, then all tests are generated manually

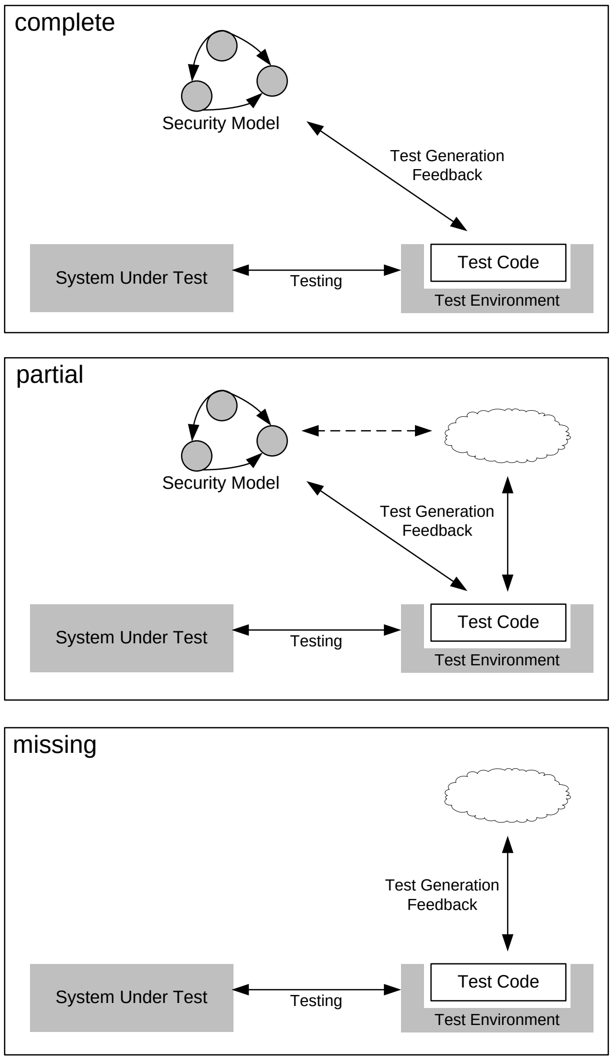


Fig. 1. Complete, Partial, and Missing Security Model

which is depicted by a cloud only. Assuming that the security model is provided, the skills needed by a test designer increase from complete to missing.

Partial security test generation is very promising because it does not require complete models and integrates the expertise of security testers. But partial security test generation is hardly employed in practice. Additionally, actual MBST approaches do not consider risk values for the test case generation. To increase the acceptance of MBST approaches in industry, and to consider existing risk-based security testing approaches, we extend the view on model-based security testing. In a novel classification of MBST approaches, we incorporate risks and partial test generation. Today, especially testing negative requirements strongly depends on expertise and experience [12]. We intend to lower the required level of expertise needed for security testing by defining new approaches based on our classification.

III. Model-Based Security Testing Classification

As motivated before, in this section, we classify model-based security tests along the two dimensions *automated test generation* and *risk*. The automated test generation dimension describes how much of the system and the security requirements is captured by formal models. The fewer information on system and security requirements is available, the more individual knowledge by the test engineer is needed to specify meaningful test cases. Fully automated test generation is only possible with formal and complete models which are typically not available. However, a possibility to support the test engineer in test design is the consideration of risk models, on the one hand, for deriving test cases, and, on the other hand, for prioritizing test execution. This results in the classification of model-based security testing approaches shown in Figure 2.

The degree of "automated security test generation" can be *complete*, i.e., all security tests are derived from a model, *partial*, i.e., some security tests are generated from a model, and others are added manually, or *missing*, i.e., all tests are defined manually. Assuming that the security model is provided, needed test design skills increase from *complete* to *missing* test generation. The more complete a security model is, the more functional are the security requirements. Note that the degree of automated test generation may vary for different system components.

The dimension "risk" can have the values *integrated* into the model or *not integrated* into the model. Note that the boundaries between the different characteristics are fuzzy. In the following sections we provide examples for each category.

A. Individual Knowledge

If a model does not support automated test generation and does not consider risk values, then individual knowledge determines the design of security tests and the selection of appropriate functional [11] and penetration testing techniques [17]. The efficiency of such techniques heavily relies on the experience and the specific domain knowledge of the test designers. However, employing the idea of pattern based testing allows to, although no better than very rudimentary, tweak this otherwise quite random testing techniques in terms of testing process itself.

B. Adapted RBT

If automated test generation is not possible because of a missing security model but risks have been evaluated, then a prioritization of tests is possible. Most actual risk-based testing approaches, e.g., [13] assign risk values to design elements and can therefore be categorized as risk enhanced model-based testing approaches without automated test generation. Wysopal et al. [14] define an adapted risk-based security testing approach based on threat models. The risk values in case of adapted RBT approaches are often only defined as additional informal artifacts, e.g., in spreadsheets. But the risk assessment itself is typically systematized in such approaches [31].

C. Scenario-Based MBT

The partial automated generation is supported e.g., by the Telling TestStories approach [32]. Telling TestStories supports the automated generation but also the manual definition of so called test stories, i.e., test scenarios modeled as UML activity diagrams or UML sequence diagrams plus assigned test data in a tabular form. Telling TestStories has been applied to model security tests of service-centric systems [33]. However, the idea of scenario-based MBT is not only restricted on UML diagrams, also the application of control flow graphs allows to derive scenarios by employing graph coverage criteria, geared towards covering high security sensitive execution traces. Obviously, following such an approach the graph replaces the classical perception of a software model, based on notions of UML. Tuglular et al. [34] suggest an approach to firewall testing based on Event Sequence Graphs, used to directly generate test cases.

D. Risk Enhanced Scenario-Based MBT

The Telling TestStories approach mentioned before is scalable for integrating risks into the test model to define a risk enhanced partial test generation process. We consider the integration of risks into Telling TestStories as future work.

E. Adapted MBT

The adapted model-based testing for security models is supported by the automated test generation approaches for access control policies discussed in Section II, e.g., [23], [26], [27], [29], which automatically generate test cases but do not consider risk values.

However, the application of adapted MBT in the field is quite rare, as often a *complete* model, as required by MBT approaches, does not exist.

F. Automated RB Security Test Generation

RiteDAP [35] is a model-based approach to risk-based system testing that employs annotated UML activity diagrams for automated test case generation and prioritization. RiteDAP is therefore an approach that manages risks on the model level and supports the complete automated test generation. But it does so far not consider the automated test generation of security tests.

The approach, suggested by Zech [36] actually attempts to go one step further as RiteDAP by additionally supporting the automated generation of test cases. Based on attack patterns and threat profiles, enabling the generation of a risk model from a system model, test case are generated out of misuse cases, automatically derived from the aforementioned risk model. Hence, this approach can be considered as an approach to automated RB security test generation, based on a tailored security model. Again, such an approach, building on a *complete* model currently lacks acceptance in the field, as most of the time, as already mentioned before, a complete

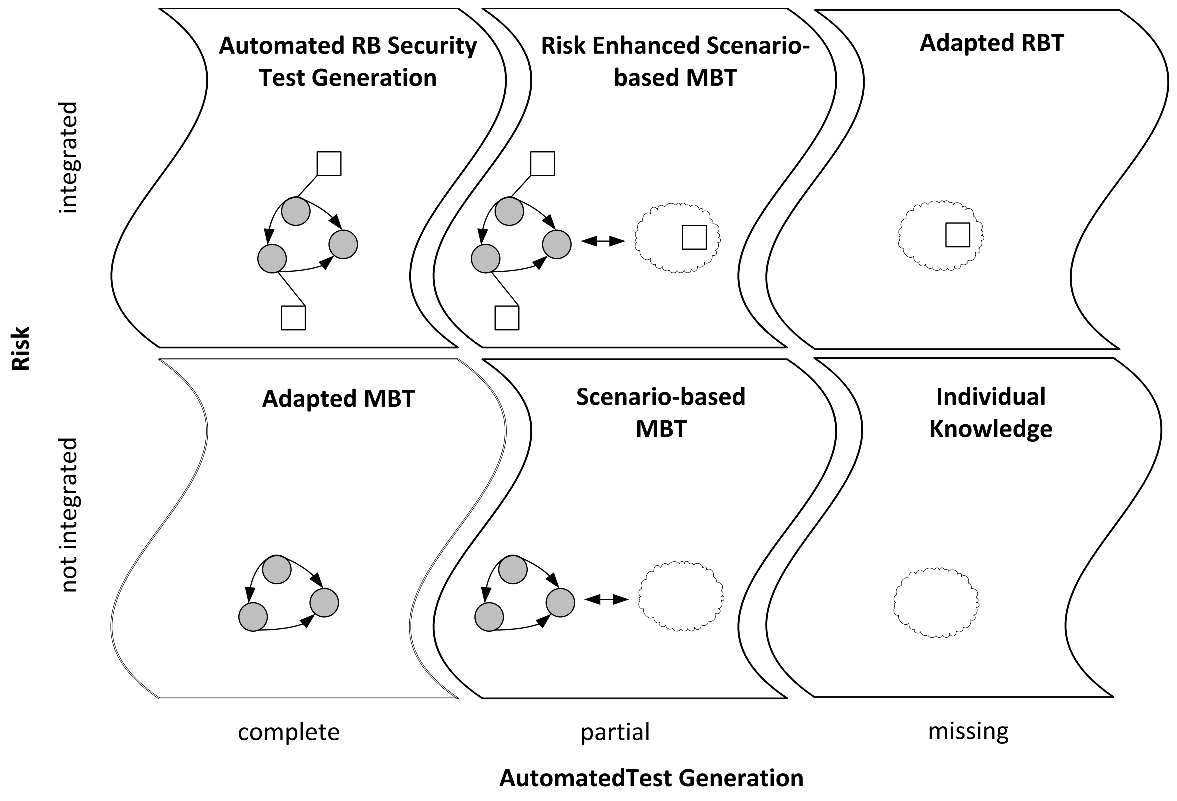


Fig. 2. Model-Based Security Testing Classification

model, capable of being employed for test generation, simply does no exist.

According to the presented categorization there exist only very few model-based security testing approaches that integrate complete or partial automated test generation and risks. Both, a risk enhanced scenario-based MBT approach and an automated RB security test generation approach contribute to utilize the full potential of model-based security testing of positive and negative security requirements. We therefore motivate the further investigation and development of risk-aware and automated MBST approaches based on existing frameworks such as Telling TestStories or RiteDAP.

IV. Concluding Remarks

In this contribution, we have first explained the benefits of model-based testing, which is already widely used today. To employ model-based approaches for testing security requirements, security models are needed which describe how a system is supposed to behave. The security model lowers the level of required security expertise of the test designer due to several reasons.

First, such a security test model improves the test designer's understanding of the software's security aspects which results in more efficient test cases. By using models, the level of abstraction is raised which enables more people to design tests. Finally, the model can be employed to automatically generate test cases. Additionally, security models are often created in conjunction with a risk analysis. This risk information can on the one hand be used for deriving test cases, and on the other hand for prioritizing test execution. We have developed a categorization along these two dimensions and provided examples for each category.

Based on an overview of security testing approaches, we have motivated the increasing importance of model-based security testing. We especially pointed out that the integration of risks into test models has not been investigated in detail, but has high potential for practical application in security testing.

According to our classification and existing security testing approaches we identify the following future research tasks in the area of model-based security testing:

- Development of a risk enhanced scenario- and model-based security testing approach on top of a scenario-based MBT approach.
- Development of an automated risk-based security test generation approach grounded on a model-based approach to risk-based testing.
- Integration of manually, semi-automatically and automatically determined metrics for the assessment of risks values in the risk model which is the basis for the integration of risks into a test model.
- Application and evaluation of model-based security testing for service-centric systems such as service-oriented architectures or cloud applications.

Acknowledgements

This work is sponsored by the SecureChange project (EU grant number ICT-FET-231101), the MATE project (FWF project number P17380), and QE LaB - Living Models for Open Systems (FFG 822740).

References

[1] D. Firesmith, "Engineering Security Requirements," *Journal of Object Technology*, vol. 2, no. 1, 2003.

[2] Common Criteria Recognition Arrangement, "Common Criteria for Information Technology Security Evaluation," 2009, http://www.commoncriteriaportal.org/thecc.html [accessed: April 7, 2011].

[3] Committee on National Security Systems, "National Information Assurance Glossary," CNSS, Tech. Rep., 2006.

[4] G. Canfora and M. D. Penta, "Testing Services and Service-Centric Systems: Challenges and Opportunities," *IT Professional*, vol. 8, pp. 10–17, 2006.

[5] T. Roßner, C. Brandes, H. Götz, and M. Winter, *Basiswissen Modellbasierter Test*. dpunkt Verlag, 2010, in German.

[6] H. Götz, M. Nickolaus, T. Roßner, and K. Salomon, *iX Studie Modellbasiertes Testen*. Heise Zeitschriften Verlag, 2009, in German.

[7] T. Mouelhi, "Testing and Modeling Security Mechanisms in Web Applications," Ph.D. dissertation, RSM, University of Rennes, 2010.

[8] W. Stallings, *Network Security Essentials*. Prentice Hall, 2002.

[9] T. Jaeger, *Operating System Security*. Morgan & Claypool, 2008.

[10] I. Alexander, "Misuse cases: Use cases with hostile intent," *Software, IEEE*, vol. 20, no. 1, pp. 58–66, 2002.

[11] M. C. C. and R. Will, "Risk–based and Functional Security Testing," Cigital, Tech. Rep., 2009, https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/testing/255-BSI.pdf [accessed: April 7, 2011].

[12] B. Potter and G. McGraw, "Software Security Testing," *IEEE Security & Privacy*, 2004.

[13] S. Amland, "Risk-based testing: : Risk analysis fundamentals and metrics for software testing including a financial application case study," *Journal of Systems and Software*, vol. 53, no. 3, pp. 287–295, 2000.

[14] C. Wysopal, L. Nelson, D. D. Zovi, and E. Dustin, *The Art of Software Security Testing*. Addision–Wesley, 2006.

[15] D. Firesmith, "Security use cases," *Journal of Object Technology*, vol. 2, no. 1, pp. 53–64, 2003.

[16] M. Bishop, "About Penetration Testing," *IEEE Security & Privacy*, vol. 5, no. 6, 2007.

[17] SearchNetworking.com, "Penetration Testing Strategies," 2011, http://searchnetworking.techtarget.com/tutorial/Penetration-testing-strategies [accessed: April 7, 2011].

[18] K. van Wyk, "Penetration Testing Tools," 2008, available at https://buildsecurityin.us-cert.gov/bsi/articles/tools/penetration/657-BSI.pdf [accessed: April 7, 2011].

[19] P. Herzog, *The Open Source Security Testing Methodology Manual 3*, 2010, http://www.isecom.org/mirror/OSSTMM.3.pdf.

[20] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, 1st ed. Norwood, MA, USA: Artech House, Inc., 2008.

[21] S. Taber, C. Schanes, C. Hlauschek, F. Fankhauser, and T. Grechenig, "Automated Security Test Approach for SIP-based VoIP Softphones," *Advances in System Testing and Validation Lifecycle, International Conference on*, vol. 0, pp. 114–119, 2010.

[22] Y. Yang, H. Zhang, M. Pan, J. Yang, F. He, and Z. Li, "A model-based fuzz framework to the security testing of tcg software stack implementations," in *Proceedings of the 2009 International Conference on Multimedia Information Networking and Security - Volume 01*, ser. MINES '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 149–152. [Online]. Available: http://dx.doi.org/10.1109/MINES.2009.111

[23] J. Julliand, P.-A. Masson, and R. Tissot, "Generating Security Tests in Addition to Functional Tests," in *AST '08: Proceedings of the 3rd international workshop on Automation of software test*. ACM, 2008.

[24] K. Lano, *The B language and method: a guide to practical formal development*. Springer-Verlag New York, 1996.

[25] P.-A. Masson, M.-L. Potet, J. Julliand, R. Tissot, G. Debois, B. Legeard, B. Chetali, F. Bouquet, E. Jaffuel, L. Van Aertrick, J. Andronick, and A. Haddad, "An access control model based testing approach for smart card applications: Results of the POSÉ project," *JIAS, Journal of Information Assurance and Security*, vol. 5, no. 1, pp. 335–351, 2010.

[26] A. Pretschner, T. Mouelhi, and Y. Le Traon, "Model-Based Tests for Access Control Policies," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, 2008.

[27] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Tutorial software quality assurance: a practical approach*, 1985.

[28] Y. L. Traon, T. Mouelhi, and B. Baudry, "Testing Security Policies: Going Beyond Functional Testing," in *The 18th IEEE International Symposium on Software Reliability*, 2007, pp. 93–102.

[29] J. Jürjens, "Model–based Security Testing Using UMLsec," *Electron. Notes Theor. Comput. Sci.*, vol. 220, no. 1, 2008.

[30] ——, "UMLsec: Extending UML for Secure Systems Development," in *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*. Springer-Verlag, 2002.

[31] E. Veenendaal, "Practical Risk-Based Testing PRoduct RIsk Management: the PRISMA method," Improve Quality Services BV, Tech. Rep., 2006.

[32] M. Felderer, P. Zech, F. Fiedler, and R. Breu, "A Tool-based methodology for System Testing of Service-oriented systems," in *VALID 2010*, 2010, pp. 108–113.

[33] M. Felderer, B. Agreiter, and R. Breu, "Security Testing by Telling TestStories," in *Modellierung 2010*, 2010.

[34] T. Tuglular, O. Kaya, C. A. Muftuoglu, and F. Belli, "Directed Acyclic Graph Modeling of Security Policies for Firewall Testing," *Secure System Integration and Reliability Improvement*, vol. 0, pp. 393–398, 2009.

[35] H. Stallbaum, A. Metzger, and K. Pohl, "An automated technique for risk-based test case generation and prioritization," in *Proceedings of the 3rd international workshop on Automation of software test*, 2008, pp. 67–70.

[36] P. Zech, "Risk–Based Security Testing in Cloud Computing Environments," *2011 Fourth IEEE International Conference on Software Testing, Verification, and Validation*, pp. 411–414, 2011.

Utilizing Domain-Specific Modelling for Software Testing

Olli-Pekka Puolitaival, Teemu Kanstrén
VTT Technical Research Centre of Finland
Oulu, Finland
{olli-pekka.puolitaival, teemu.kanstren}@vtt.fi

Veli-Matti Rytky, Asmo Saarela
Elektrobit Wireless
Oulu, Finland
{veli-matti.rytky, asmo.saarela}@elektrobit.com

Abstract—**Automated execution of manually defined regression tests is a very widely used and well-known area. While test execution can be more easily automated, test case creation and maintenance are still mainly manual efforts and practically the biggest cost factors in software testing. We view writing test cases as basically a programming activity and believe it can thus benefit from extended application of generic programming tools and techniques. In this paper, we describe our work in applying domain-specific modelling (DSM) to the domain of test case creation. DSM is a variability handling method typically applied in software development. It is widely used and powerful method best applied when there are several kinds of variations. DSM is typically tailored to make own optimized modelling solution inside a company, after which it can be applied effectively and without requiring specific programming skills. In this paper we describe how we have applied DSM to describe variability in software behaviour in terms of test cases, and its application in a case study. The results show a reduction in the cost of over test automation.**

Keywords-domain-specific modellin; test automation

I. Introduction

Test automation these days is a popular concept with an extensive body of knowledge and a large set of mature tools available. The most popular approach in this domain is the automation of test case execution [1]. Automated execution of test cases gives a lot of benefits such as faster execution times, automatic smoke tests after each commit and nightly regression tests. Test cases are typically written in a form of programming language, describing input- and output sequences, data values, and their expected interactions. These test cases are then executed using a test automation framework. This activity of test execution has a long history and a large set of mature testing tools and techniques available. While test execution can be viewed as highly advanced, the largest effort in the software testing process is in manual test case creation and maintenance. As we view test case creation as closely related to general programming activities, we believe it is possible to use more advanced techniques and tools from the domain of software engineering to also enhance the test creation activity.

One way to provide more effective support for test creation and maintenance is to use a higher abstraction level for describing the test cases. In the software engineering domain this is commonly addressed through different modelling techniques. A specific approach for this in the software engineering domain is domain-specific modelling (DSM). In DSM a specific optimized modelling solution is first tailored for a chosen domain by the domain expert and then applied continuously by the domain users [2].

Traditionally, DSM is used to handle variability of product lines. In this case, the different products in a product line are modelled using a common notation and focusing on the differentiating aspects with the optimized, domain-specific modelling language. In our view, this translates very effectively to the domain of test automation and specifically that of test creation.

In software testing, we typically need to exercise the different aspects of system behaviour with different variations. For example, when we have one boundary value that needs to be covered, we need at least three test cases to cover that (below, equal, and above boundary values). Therefore, we view software testing as a domain with high variability and large potential to benefit from the different aspects of DSM. Another related aspect related to this is Model-Based Testing (MBT) [3] that aims to improve test coverage by automatically generating test sets from a system behavioural model utilizing several algorithms. In our previous work, we have described how DSM can be combined to provide added benefits for MBT [4]. However, although advanced test generation techniques such as MBT can be powerful, our experience is that there is still need for manually defined test cases. In this paper, we present our work on using DSM as an aid for more effective creation of test cases. We demonstrate this with the aid of a case study, including the observed cost-benefits achieved.

The rest of the paper is structured as follows. In Section II, we describe the background concepts relevant to the work presented in this paper. In Section III, we describe the case study system used to illustrate the discussed concepts through the rest of this paper. In Section IV, we describe our approach to using DSM to help in test creation. In Section V, we describe the results of the case study and discuss the DSM test modelling approach a wider context. Finally, conclusions summarize the paper.

II. Background Concepts

In this section, we describe the background information required to understand the concepts described in this paper.

A. *Test case automatic execution*

With test case automatic execution we refer to a tool chain in which test cases are described in a form of programming language, such as a scripting language, and tests can be run automatically once they have been specified (as defined, e.g., in [1]). This tool chain needs to address the different needs for test components in test automation. This includes providing test input as stimuli for the system under test (SUT), test output as a reference of the expected response, and a test harness to link the test cases themselves to the SUT. A test oracle is a component needed for determining the correctness of the behaviour of software during (test) execution [5]. Together these components form what we term as the test execution environment.

Test automation in this type of an environment takes the following process. First the test scripts are written (typically manually) by a test expert. This includes defining both the test input and expected output in the given notation for the test execution environment. These tests are then executed using the test execution environment and the results are presented to the user.

B. *Modelling for testing purposes*

Modelling for testing purposes can be divided in two categories: modelling test cases, and modelling for test generation. Modelling for test generation refers to creating models that are used as a basis for test generation with techniques such as model-based testing. We have discussed this aspect before in [4], and in this paper we focus on the test case modelling aspects.

Test case modelling refers to modelling specific test cases separately in terms of chosen abstraction level. This can be either textual, graphical or some hybrid notation. In this paper we discuss this in terms of DSM concepts, which are in our case graphical or hybrid notations. Some of the most popular existing graphical notations for test modelling are UML testing profile [6] and TTCN-3 graphical presentation format [7]. These and other generic languages can be widely applied but are not as powerful for the chosen domain as the DSM based notations. In this paper, we describe a case study in using test modelling using DSM tools and concepts.

C. *Domain-Specific Modelling*

DSM is about creating a new modelling language based on domain concepts and using a (typically) self-made code generator to transform this to a different form such as textual source code. These modelling languages are typically easier and quicker to understand for most of the people because they describe the intended domain using higher level domain concepts. These languages can be visual, textual or combinations of both. The modelling language is typically constrained to reduce the modelling options to only those relevant to the expected variance in the domain, which also serves to simplify the modelling process and reduce the number of errors in generated output. Based on our experiences, the DSM are most useful when there is some kind of variability in the product.

The idea in addressing variability in the DSM language is that static part of output (the common part of the target domain) is in generator or in the used platform. Thus the modelling can focus only on the varying aspects of the domain, while the static parts are provided off-the-shelf. Because of this, the support for the dynamic parts can be highly optimized, simplified and made easier to use. A common application domain is with product lines due to variability between products [8] [9].

Domain-Specific Modelling work flow is following:

1. Create a modelling language based on your domain concepts
2. Write a generator which generates your code
3. Create a model using your language
4. Generate the code or document or what you want

Because the code is generated from a model, the debugging can be also made in to the model. Normally the system sends information from its state and the workbench highlighted it to the model. If an error exist it is easy to see in which part of model it happens. There can also be a need to display additional debug data into the model, e.g., performance metrics.

D. *Generating test cases utilizing DSM*

As test cases are typically expressed as scripts using a textual notation, this makes their generation from specific domain test models a viable approach. We can summarize that the main benefits of DSM in the context of test automation are:

- Model is easier to understand because it is expressed using our domain concepts.
- Modelling is faster because it is optimized for the domain and constrained to avoid obvious mistakes.
- Models can be expressed visually, providing for easier to understand test expressions.
- Non-programmer can create test cases.

In this case, DSM can be understood also as a more illustrative user interface for test scripts or a test script visualization method. The structure of DSM for test case generation is illustrated in Figure **1**.

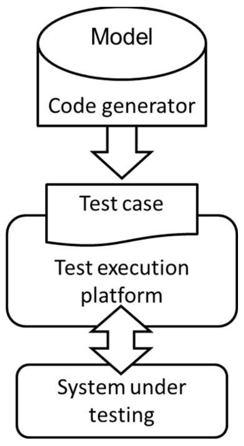


Figure 1. Domain-specific modelling with automated test execution

III. MILITARY PHONE AS TESTING TARGET

The case study system described in this paper is using Elektrobit Tough Voip (ETV) [10] as the SUT. ETV is a military Voice Over Internet Protocol (VOIP) communication device. The device has to be very easy to use and resistive because it is made for military purposes. As failures and bugs in a military system can obviously lead to big problems, the quality and reliability of these devices are a major concern. ETV is presented in Figure **2**.

Figure 2. EB Tough VOIP

The ETV has point to “point call” and “all call” features. In a practical deployment setting, all devices have to be connected on the same network and any terminal can connect to another one just by dialling the number of that terminal. This is the “point call” feature. The “all call” feature connects a single device to the all other devices in the network using a specific dialling pattern.

Our viewpoint in testing these devices is that of a high-level abstract black-box viewpoint. From this perspective, the complexity is not in the functional on one particular device, as a single device does not contain highly complex external behaviour. The main complexity is in having many devices connected to the network, calling each other within very tight time limits and in varying sequences. While completely manual test execution is possible, without automation it is hard to address fast time limits or to create test cases for time border values. Therefore, the system also has a TTCN-3 based testing environment for executing automated test cases against the devices.

Besides fully manual test execution, also test creation and maintenance even with an automated test execution environment has its own issues. As noted before, there is a requirement to test various connection- and call-sequences as well as various time limits within these sequences. With the traditional TTCN-3 test scripting and environment, the maintaining of the test suite was found to require a large effort and to require a large amount of TTCN-3 expertise which was found expensive and limited in availability. To address these issues, we created a DSM based solution where the test cases can be expressed in domain concepts without having to work with detailed internals of the TTCN-3 notation.

IV. A MODELLING LANGUAGE FOR TEST CASE GENERATION

In creating a domain-specific modelling language for ETV testing, we used the main domain concepts as a basis. These are the devices themselves, the call types, their ordering and time constraints. The created language is composed of the basic test automation elements of test suite, test case, and test setup. The test suite model is a collection of test cases, the test case model describes test case structure for a single test case, and test setup describes the setup of the device under testing. The overall test model architecture is illustrated in Figure **3**.

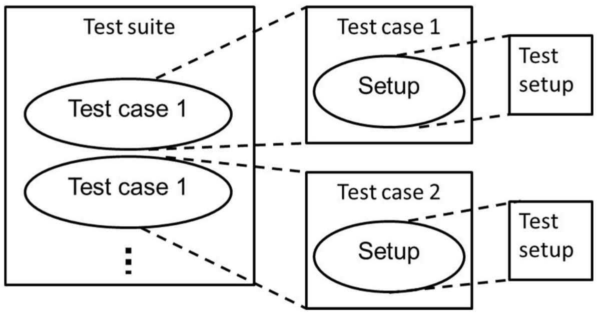


Figure 3. Test model architecture

We used MetaEdit+ [11] as a language workbench for creating our DSM language, and TTCN-3 as the test scripting language. The effort in creating the overall test automation environment was shared with two engineers. The first engineer developed the modelling language (modelling language developer) and the associated TTCN-3 script generator, and the second one manually developed TTCN-3 based test scripts and the overall test case execution process (test case developer).

At the beginning, the test case developer gave a test script sample and initial requirements of modelling language to the modelling language developer. The modelling language developer used these as a basis for creating the first DSM language version. This and the associated test script generator were then evaluated by the test case developer, who made change requests based on the results. Based on this feedback, the modelling language developer made fixes to the language and generator. After one week of iterations, the language and test execution environment was ready for testing in a real test environment.

In the following subsections we describe each of the model elements and an additional test run visualizer component used to show the actual executing test cases.

A. *Test suite model*

The test suite model is a collection of test cases. Test cases are represented as objects in the model and the colour of these objects tells their enabled status. A green object is enabled and red ones are disabled. Figure **4** shows an example of a test suite model, where two of the test cases are enabled and one is disabled.

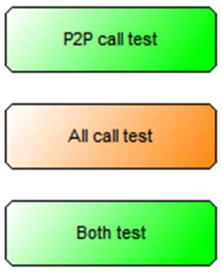


Figure 4. Test suite graph

The test case objects in this model contain sub-models that can be opened to further explore the test case represented by that model. The test case name is presented as the sub model name in the visual presentation shown in Figure **4**. This example has three test cases, one for “point call”, one for “all call”, and one for testing both. The both test graph is presented in the following section.

B. *Test case model*

A test case model is used for representing individual test cases for the EVP system. This model consists of six different types of objects and two types of connection lines. These components are the following:

- *Start object* represents the test case starting point.
- *End object* represents the end of test case.
- *Call object* represents a test step where a device calls to another device or devices including oracle disabling option if time limits are too tight for oracle between the call and next call.
- *End calls object* represent a test step where a single device ends the call.
- *Device object* represents a device including device name and phone number.
- *Test setup object* is a link to the test setup graph.
- *Connection lines* are arrows in to the model representing test case order, calling device and destination device.

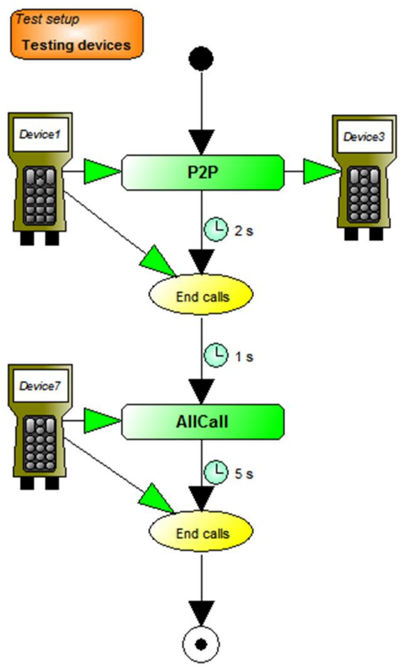


Figure 5. Test case graph

Figure 5 shows an example test case using this notation. It describes the following scenario:

1. Device1 makes a call to Device3
2. After 2 seconds Device 1 ends the call
3. After 1 second Device7 calls to all devices
4. After 5 seconds Device7 ends the call

C. *Test setup model*

A test setup model represents devices in the current test network. In practice, the test setup changes a lot and we do not wish to change all the test cases in case the setup needs to be changed. Instead, we wanted an option that allows us to change the test setup and immediately re-run the existing test cases with a different set of devices and their configurations. In our modeling language, the test setup model is used to represent this type of information and to allow the modification of the different test setups independently of the individual test cases.

In the tests, the test setup model is mainly used for automatic device initialization. The test oracles in the *AllCall* test steps also use the test setup model because they needs to verify the status of all devices that are connected to the network. Figure **6** shows an example of a test setup model in our case study.

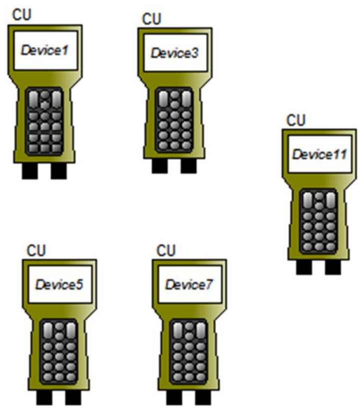


Figure 6. Test setup graph

D. *Test execution visualization*

Showing the execution of test cases is always important for understanding the execution and debugging the results. Initially, our test execution process illustration was just a lot of text running quickly in the command line shell on the test execution platform, which in our case was called Elektrobit Test Tool Platform. This was not a very human friendly form of feedback.

To provide a better support for visualization of test execution, we created visualizations of the DSM test models, showing at all times the current object of the testing as highlighted. This visualization was made using application programming interface (API) of the MetaEdit+ DSM tool, which allows connecting external elements to the model code using the commonly supported SOAP [12] protocol. Practically, the test environment would send SOAP messages to the MetaEdit++ tool, where the specific test DSM language and model received these messages and as a result highlighted the matching elements in the model. For us, this illustrated the possibility to easily and effectively use the test models as tools for following test execution, reporting the test results and debugging possible errors in a human friendly way.

E. *Test generation*

Test generation is made by MetaEdit+ providing MERL language. MERL is a small programming language and it is decided for code generation providing effective way to go through the model and print the output.

Before starting any generation, the generator checks the most typical errors and reports if some exist. The checks are self-made and based on our experiences. First, the generator makes initializations based on test setup model. Then the generator starts from start state object and goes through the model following the arrows and printing object specific code and finally ends to the end state object.

F. *Test model execution*

In addition to support the test creation (modelling) process, we also aimed to automate the test case execution itself. Initially, test cases were created and executed manually. In the first phases of our DSM application, we proceeded to generate the test cases for the execution environment from the models which only required manual linking of the generated test cases to the test environment. From this, we further linked the whole test environment into the modelling environment, allowing one to run all tests directly from a single interface and not requiring any direct low-level interaction with the TTCN-3 notation or test environment itself.

Finally, only pressing a single button in the test case or test suite model view is needed. After the test execution, the results are presented. The test execution consists of following steps:

1. User presses test-button in the model.
2. MetaEdit+ generates test cases using the domain-specific test code generator.
3. OpenTTCN tool compiles the test cases.
4. The test platform executes the test cases.
5. During the test case execution, the test platform sends execution information to the MetaEdit+ tool that visualizes the execution in the model.
6. After a test ends, the OpenTTCN provides a test report with highlighted issues that were observed while test execution.
7. Tests engineer check the report, fix from test model and generator or report system bugs to the developers and starts over the testing process.

V. Results and Discussion

Our initial goal in creation of our DSM based test case modelling approach was to enable test case creation for users without requiring specific detailed knowledge of the test environment or the used TTCN-3 notation. However, in practice as experienced in our case study, the test expert still defined most of the test models. This is a person who had also previously been involved in test script creation and manual test case creation for the SUT using the TTCN-3 notation. Thus the test expert already was familiar with the underlying notations and had expertise on the expected SUT behaviour.

However, despite this background we found our results very encouraging. Our experiences are based on about twenty models in real environment and those models are more complex and having some more features than case study models. The test engineer estimated the modelling using our new DSM approach as being at least ten times faster than manual test script writing. The set-up time for creating the modelling notation and the test script generator for this notation was about one week. We used one more week for adding more advanced features and for making our system mature. It took about one week to make changes for the underlying TTCN-3 code to make it easier to generate and to fix bugs we found in it. However, this work was found to improve the overall test system and was not just useful only for our DSM approach. Since our DSM approach needed to evolve to adapt to the actual needs, we added several features to the language. The average time for integrating a new feature was about 30 minutes. In addition, we extended the language to cover other variations of EB Tough VoIP, which took about 3 days. These results were available as we recorded the time we took in the different steps and the company in the case study had also made effort to record this information for their previous approaches.

In our case study, we found that the test case development speed is not the only benefit. The developers and other interest groups assessed the DSM based test case creation and visualization approach as easier to understand. As the test case is graphical, it is easy to see what it does. During the test execution, the progresses are visualized. This saves time in analysing test cases.

In our experience, the development of DSM based test script generation is like normal system development. The DSM method is different but it is just one technology to learn. The DSM modelling workbenches (such as MetaEdit+) are just one form of a programming environment. As these are created specifically to support building this type of modelling notations and environments, the work amount for use has been in the amount of weeks instead of months or years.

The main challenges for applying DSM for test creation are in creating a good modelling language. This requires good understanding of both the application domain and the test automation domain. In this case the modelling language creator and maintainer needs to have quite a wide understanding of the system to create a suitable and powerful test modelling environment. However, this is offset by the reduced need in the test modelling phase where less detailed knowledge of the low-level operations is needed.

Our development style for the case study described in this paper was close to consulting. An external research entity was helping an industrial partner build a DSM test modelling approach and apply it. Thus the people creating the test modelling language and writing the test scripting environment were different. This approach is perhaps not as effective as when entity person is creating both the modelling language and the test scripting environment. However, our experience is that the result is better because of discussion with different entities. In case of a single company this can also be different people from different entities inside the company. During the development people from both entities were describing the solutions to each other and getting feedback on their part. This helped avoid overly complex structures and modelling approaches, as the other entity had to use the results of the other entity in both cases and immediately objected if they found the result too complex for easy adoption.

While results might be different in a different type of an environment and when applied by people with different backgrounds, we believe our result can encourage other people to try our DSM based test modelling and creation approach.

VI. CONCLUSION

In this paper, we described how DSM languages can be used to support the manual test case modelling and creation process, which we observed as one of the most expensive parts in software testing. Using a practical case study, we illustrated the approach in practice. In our experience, the results have been encouraging. The setup effort to create the required modelling languages and environment using existing DSM tools was a couple of weeks and provided more than ten times faster test case creation speed in comparison to previous experience in the case study environment. Thus we can conclude that our modelling approach takes more investment in the beginning but quickly becomes more effective as more test cases need to added and existing ones need to be maintained.

In the future research we need to try this approach in several cases in different domains and get more experiences in its application. We are also using model based testing with DSM test models for reaching even more enhanced test automation.

VII. REFERENCES

[1] Dustin Elfriede, Rashka Jeff, and Paul John, “Automated Software Testing,” Massachusetts: Addison Wesley Longman, 1999.

[2] Kelly Steven and Tolvanen Juha-Pekka, “Domain-Specific Modeling,” New Jersey: John Wiley & Sons, 2008.

[3] M. Utting and B. Legeard, “Practical Model-Based Testing: A Tools Approach,” Morgan Kaufmann, 2007.

[4] O-P. Puolitaival and T. Kanstrén, "Towards Flexible and Efficient Model-Based Testing, Utilizing Domain-Specific Modelling," in 10th Workshop on Domain Specific Modelling, 2010.

[5] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley, "Specification-Based Test Oracles for Reactive Systems," in Proc. of the 14th Internation Conference on Software Engineering, Melbourne, Australia, 1992, pp. 105-118.

[6] OMG. UML Testing Profile. [Online]. http://utp.omg.org/ 15.7.2011

[7] ETSI. TTCN-3 Graphical presentation Format. [Online]. http://www.ttcn-3.org/StandardSuite.htm 15.7.2011

[8] Mika Karaila, Domain-specific Template-based Visual Language and Tools for Automation Industry. Tampere: Tampere University of Technology, 2010.

[9] Kärnä Juha, Tolvanen Juha-Pekka, and Kelly Steven, "Evaluating the Use of Domain-Specific Modeling in Practice," in 9th OOPSLA Workshop on Domain-Specific Modeling (DSM 2009), Orlando, 2009.

[10] Elektrobit wireless. EB Tough Voip. [Online]. http://www.elektrobit.com/what_we_deliver/wireless_solutions/device/products/eb_tough_voip 15.7.2011

[11] Metacase. MetaEdit+ Modeler- Support Your Modeling Language. [Online]. http://www.metacase.com/mep/ 15.7.2011

[12] W3C. (2007, April) SOAP Version 1.2. [Online]. http://www.w3.org/TR/soap12-part1/ 15.7.2011

Comparison of off-chip interconnect validation to field failures

David Blankenbeckler, Adam Norman
Intel Corporation
Santa Clara, CA, USA
David.Blankenbeckler@Intel.com
Adam.j.Norman@Intel.com

Michael Shepherd
Dell Inc.
Round Rock, TX, USA
Michael_Shepherd@Dell.com

***Abstract*— Memory subsystem errors continue to be a common problem in modern computer systems. Through a large scale field study, this paper will introduce the interconnect transient margin validation metrics and compare to the observed field failures. The results will demonstrate that transient bus errors are not a dominant cause of system memory problems.**

Keywords – DDR, DRAM, memory, bus margin

I. Introduction

Memory subsystem errors have remained a common form of failure since the advent of the computer. While much work has been done to reduce failures and gracefully handle them, they continue to be a significant problem in modern day computer architecture.

Over the past several years, at least two large scale studies have been conducted to help quantify the extent of memory bus related failures. The recent white paper “Dram Errors in the Wild: A Large-Scale Field Study” by Bianca Schroeder, et al, indicated the rate was as high as 1/3 of systems experiencing at least one memory error per year [1]. Another study found at least 11 systems out of 212 that show symptoms of memory errors [7]. But what is the cause of these high failure rates? Most large scale studies have focused on Soft Error Rates (SERs) due to alpha particles [8], junction/cell leakage, manufacturing defects and the rate of errors across die shrinks.

During the late 70’s, alpha particles from decaying package contaminants were a dominant source of memory errors [2]. Around the same time, researchers at IBM found that cosmic rays were also a source of transient memory errors, even at sea level [3][4]. In one study it was reported that memory errors were about 100x more likely at the altitude typically used by commercial aircraft [5]. These radiation induced errors are generally referred to as soft errors and have been the subject of much research. Today, this phenomenon is generally understood and thus effective mitigation techniques have been and continue to be developed [6].

Besides soft errors, there are various types of hard errors which could occur in either the memory controller or DRAM device. The most common hard failures are due to defects produced during the wafer manufacturing process. These failures may also be caused by design marginalities or aging effects.

This study is uniquely different from prior work in that the goal was to better understand the relationship between bus related margins and their resulting error rates. As bus speeds have increased signal integrity has become a factor suspected of being a significant contributor to transient errors. A properly designed system should have a reliable interface between the memory controller and the DRAMs. However, in the real world, factors such as excessive manufacturing variation or unexpected environmental conditions may impact reliable data transfers. Over large volume, these variations may increase noise on the bus and thereby increase the chance of transient errors. As shown in Figure 1, the year over year incident rate of end user memory errors has remained relatively unchanged. This data indicates that even across DRAM technologies and speeds between 2006 and 2009; memory system failure rates have remained relatively stable. Why do we see this consistent failure rate and what is causing it?

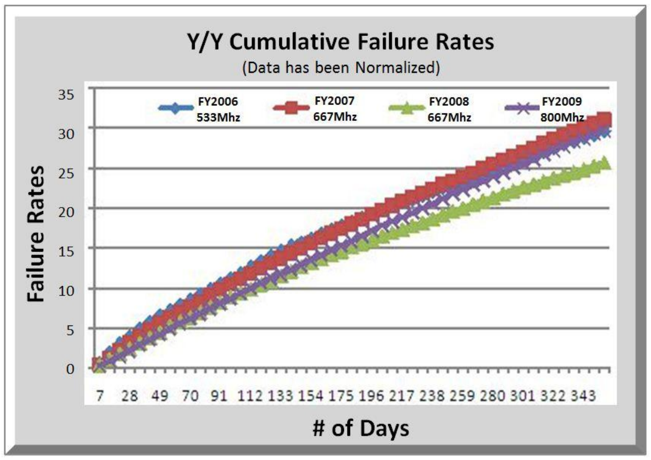


Figure 1. Year over year failure rates have remained relatively stable.

Could signal integrity now be a key factor in transient memory errors? This paper will explore this question through a large scale study comprised of nearly a quarter million systems and over a million DIMMs (dual in-line memory module). Section II will describe the concept of bus margin which provides a measure of bus noise susceptibility and the different sources of memory subsystem errors. Section III will describe the data collection methodology. This includes the measurement and collection of memory bus margin at the system assembly factory as well as the data regarding end user memory issues. Section IV provides analysis of the data leading to the conclusion that bus margin is not a dominant source of end user memory issues. Finally, Section V will summarize and conclude the paper.

II. MEMORY SUBSYSTEM FAILURE CHARACTERIZATION

There are four main sources of memory errors as shown in Figure 2. At a high level, memory subsystem errors can be categorized as:

1. Internal Memory Controller Errors include logic or timing faults inside the memory controller.
2. Internal DRAM Errors include logic faults, timing faults, and cell faults.
3. Bus errors, which occur when one device (memory controller or DRAM) transmits one state but, due to noise or other factors, the other device receives the data in the opposite state. The susceptibility to transient bus errors is commonly measured by bus voltage and timing margin.
4. Soft Errors, which refer to radiation-induced transient events whereby a bit is flipped due to interference from energy sources such as cosmic rays or alpha particles. These are random events, which are very unlikely to repeat and cause an end user DIMM replacement. The primary focus of this paper is to distinguish the relative contributions of the other three sources of memory subsystem errors. The other three sources often appear random but are usually repeatable.

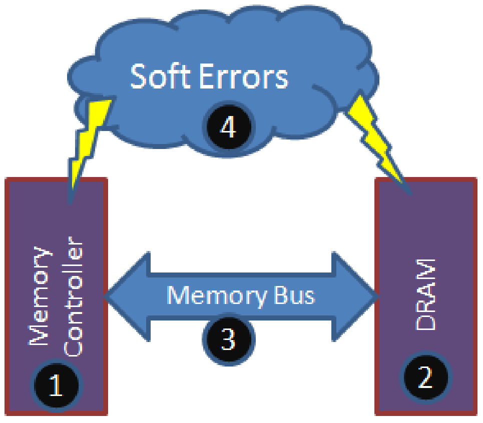


Figure 2. Sources of memory subsystem errors.

A. DDR3 Bus Margin

There are many sources of noise which can create transient errors on a high speed bus such as DDR3. These include crosstalk, inter symbol interference (ISI), and power delivery issues. In addition, there is manufacturing process variation that impacts the performance of the bus. Examples include impedance variation of the printed circuit board, variation in the nominal supply voltage, and variation in the transmitter and receiver characteristics. To account for noise and high volume manufacturing variation, system designers commonly use the concept of bus margin.

Bus margin, in concept, is a measure of the amount of noise a bus can sustain before an error is induced. For many buses, such as DDR3, this concept is implemented in practice by measuring the voltage and timing margin. The measured margin is then compared against a minimum expectation of margin, or guardband, to account for factors such as different data patterns, high volume manufacturing variation, and other factors not included in the measurement.

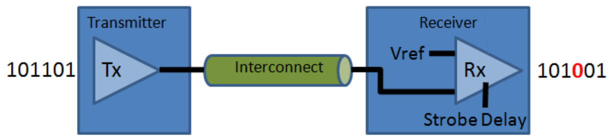


Figure 3. Data (DQ) bus topology for DDR. Vref and Strobe Delay are the bus margining offsets that can be adjusted to produce a bit error.

A typical implementation of measuring bus margin is to transfer a set of patterns over the bus while adjusting either the voltage reference (Vref) or internal timing controls to alter the relationship between the clock and the transmitted or received data, as shown in Figure 3. The voltage or timing is shifted to the point that a data error occurs. The voltage or timing offset required to induce an error establishes the voltage or timing margin for that specific configuration and conditions. Voltage and timing margin are measured for both the positive and negative direction, as shown in Figure 4. For example, the Vref is adjusted up until failure and is also adjusted down until failure. This establishes a high side and low side voltage margin and provides a source of parametric data which can be analyzed. Likewise, the sampling position (strobe delay) is moved both left and right to establish timing margin in both directions. Voltage and timing margins are measured at both ends of the bus, the memory controller in the CPU as well as the DRAM's receiver.

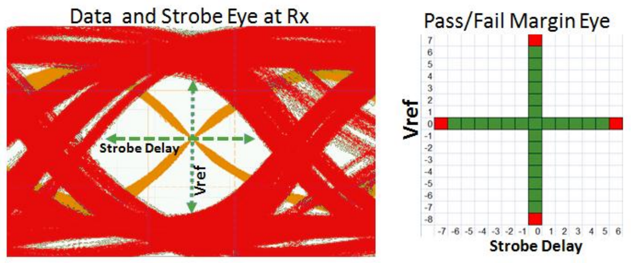


Figure 4. Bus margin example. The left plot shows the data and strobe signals at the Rx pad. The right plot shows the timing and voltage bus margin after adjusting the Vref and strobe delay until a bit error.

Bus margin is a system level metric which can be impacted by multiple factors. Specific areas that can have an impact include: varying characteristics of the transmitters drive strength for the Memory Controller and DIMM, the receivers jitter tolerance, the interconnect, the board, and the connector.

The focus of this paper is on understanding the relative impact of bus margin on the overall population of memory subsystem errors. In this study, the specific factor contributing to low bus margin it is not generally known, however we do know which margin parameter was at risk. More importantly perhaps, the data shows us those issues related to bus margin versus the other possible causes (Internal Memory Controller Error, Internal DRAM Error, or Soft Error).

III. Data Collection Methodology

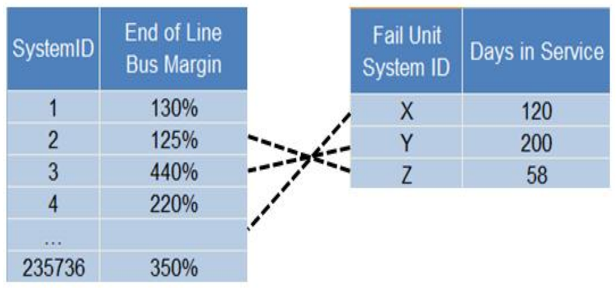
Figure 5 shows an example of the data collection methodology where a large number of server systems (235,736) were margined during the production test process and then correlated to failures at the customers' site.

A. *Factory Data Collection*

Bus margin data used in this study was captured in the system manufacturing and test process at a large server system manufacturer. Using built in test features, the margin data was collected at multiple points throughout the test and stored to a database for future analysis before the system was packaged and shipped to the end customer for installation. Margin data was collected, however, it was not used as a production pass or fail screen. In some cases, other system level tests failed and the memory modules or CPU were replaced but in those cases, the margin data was recaptured after the system was repaired. Only the final data for as-shipped configurations were used for this analysis. Consequently, the margin data in this study represents the actual margin data of the systems as they were shipped out of the factory.

B. *End User DIMM Replacements*

Tracking of end user DIMM issues was accomplished through an analysis of service call data for systems manufacturer over a period of 360 days. All systems which required a DIMM replacement in the field were identified and correlated back to the margin data collected for that system in the factory. Note that although these service calls may have included replacement of other system components in addition to memory, such as motherboard or CPU, in all cases, the DIMM was replaced.



SystemID	End of Line Bus Margin
1	130%
2	125%
3	440%
4	220%
…	
235736	350%

Fail Unit System ID	Days in Service
X	120
Y	200
Z	58

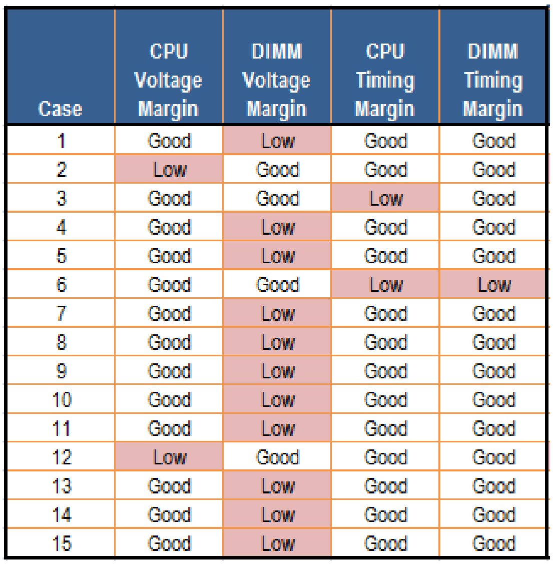
Figure 5. Data collection overview. Failing systems are cross referenced to original "end-of-line" bus margin for analysis.

IV. Data Analysis

A. *Factory Margin Distribution*

Significant insight can be obtained by analyzing the distribution of observed DDR bus margin across the resulting high volume factory dataset. The resulting data included 6 different server board designs across many different DIMM configurations. Since the data was collected in the factory environment as a study versus a screen, the margins measured represent exactly what the end customer would experience. In other words, the margins were simply measured and logged – a low margin case was shipped 'as is' to the end customer.

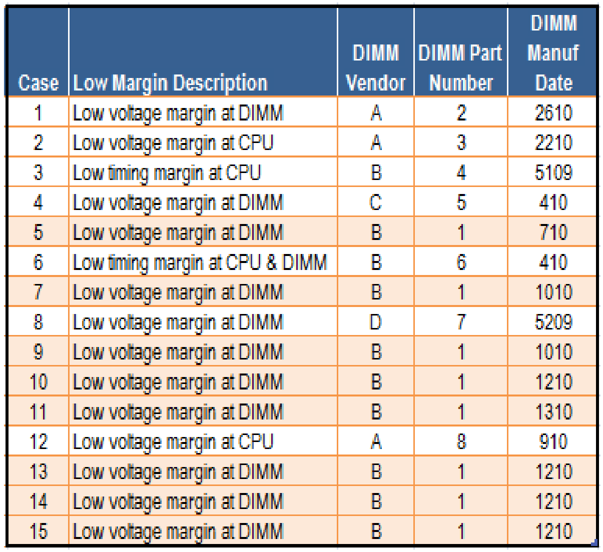
Therefore, it is interesting to consider the number of systems that fall below the minimum margin expectation (guardband). The data in Figure 6 shows that only 15 cases out of 235,736 systems were below the guardband which equals a system per million (SPM) of 64. This indicated that about 64 systems out of a million, or 0.0064%, may be at risk of experiencing a bus related error if margins remain the same over time.



Case	CPU Voltage Margin	DIMM Voltage Margin	CPU Timing Margin	DIMM Timing Margin
1	Good	Low	Good	Good
2	Low	Good	Good	Good
3	Good	Good	Low	Good
4	Good	Low	Good	Good
5	Good	Low	Good	Good
6	Good	Good	Low	Low
7	Good	Low	Good	Good
8	Good	Low	Good	Good
9	Good	Low	Good	Good
10	Good	Low	Good	Good
11	Good	Low	Good	Good
12	Low	Good	Good	Good
13	Good	Low	Good	Good
14	Good	Low	Good	Good
15	Good	Low	Good	Good

Figure 6. Cases falling below guardband

Further analysis of the data, shown in Figure 7, indicates that of the 15 systems below guardband, 8 of those were the same DIMM part number/type. These 8 DIMMs were produced in a limited DIMM manufacturing date range of 6 weeks. In fact, 6 of the 8 were in a 3 week period. This particular DIMM represented only 1.6% of the population of DIMMs yet accounted for more than half the low margin cases. This strongly suggests a manufacturing deviation or test hole in the DIMM manufacturing and test process leading to a bus marginality situation. If you were to remove this sub-population of “defective” DIMMs, the effective ratio of systems at risk of bus errors would drop to 0.0030% or 30 SPM.



Case	Low Margin Description	DIMM Vendor	DIMM Part Number	DIMM Manuf Date
1	Low voltage margin at DIMM	A	2	2610
2	Low voltage margin at CPU	A	3	2210
3	Low timing margin at CPU	B	4	5109
4	Low voltage margin at DIMM	C	5	410
5	Low voltage margin at DIMM	B	1	710
6	Low timing margin at CPU & DIMM	B	6	410
7	Low voltage margin at DIMM	B	1	1010
8	Low voltage margin at DIMM	D	7	5209
9	Low voltage margin at DIMM	B	1	1010
10	Low voltage margin at DIMM	B	1	1210
11	Low voltage margin at DIMM	B	1	1310
12	Low voltage margin at CPU	A	8	910
13	Low voltage margin at DIMM	B	1	1210
14	Low voltage margin at DIMM	B	1	1210
15	Low voltage margin at DIMM	B	1	1210

Figure 7. Low Margin Cases by DIMM Information. Highlighted rows are the same part number and date code range of week 7-12, 2010 indicating a DIMM manufacturing excursion.

Consider this low percentage of systems at risk of bus errors (0.0030%) compared to either the 30% of systems experiencing memory errors in one large scale study [1] or a more commonly expected rate of 10%-15%. Note in the referenced study [1] that these systems experiencing errors have a median number of errors between 25 and 611 per year. Given the random nature of Soft Errors, there is strong evidence that these are instead related to one of the other three sources. The margin data also suggests that relatively few systems should experience bus errors which would indicate that the bulk of end user memory errors are likely not Soft or Bus Errors, but instead either Internal Memory Controller or Internal DRAM Errors. This of course assumes that the populations of systems from the two studies were similar. We’ll explore this from another angle by looking at service call data for DIMM replacements.

B. End User DIMM Replacement Analysis

The prior analysis was done against systems that had low margin and were at risk to fail. Next we will consider systems that actually did experience some form of memory error in the field. In this analysis, we will study systems that required a DIMM module replacement at the end customer installation.

The systems which required DIMM replacement were cross-referenced back to the bus margin data collected for that specific system when it was tested at the factory. The bus margin for these systems was then compared against the minimum bus margin guardband expectation. The data in Figure 8 shows that only a small proportion of the systems requiring DIMM replacements contained low margins at the time the system was shipped from the factory. Only 0.16% of the systems were below the margin guardband and in fact, even if you double the guardband, this would still be less than 1% of systems.

Clearly, the margins on the memory bus are a minor factor driving field DIMM replacements for the population of systems under study. What is driving these replacements then? Unfortunately, detailed failure analysis was not possible for the failures returned from customer sites, but we can use the data we have to draw some important conclusions. The factory bus margin data indicates that both the CPU and DRAMs have sufficient voltage and timing margin to ensure robust data transfers. Assuming that bus margins have not degraded over time, there is a strong indication that signal integrity issues are not a major factor in memory failure rates. Considering that the sub-population of systems requiring a DIMM replacement included 32 unique DIMM part numbers across 5 different vendors, margin degradation due to aging seems unlikely. While it might be reasonable to assume that a particular DRAM design might have degradation problems due to aging, it is very unlikely that this is a widespread problem across so many different part numbers and DRAM suppliers. What about margin degradation due to aging of the CPU? The fact that the DIMM is being replaced and thereby resolving the issue contradicts this theory and indicates it is not the CPU.

Given that these DIMM failures don't correlate to low bus margins as measured in the factory and it seems unlikely that DRAM I/O degradation is a widespread problem, it is assumed that these DIMM replacements are largely driven by internal DRAM issues.

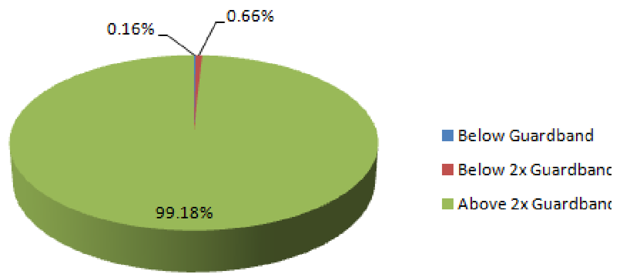


Figure 8. Chart shows the percentage of systems with low bus margin. This is for all field memory failures within our data set.

V. Conclusions

Through a large scale study of almost a quarter of a million systems and over a million DIMMs, we have found that memory bus margin is a minor contributor to memory subsystem issues which drive end user DIMM replacements. The key data supporting this conclusion:

- Prior studies indicate that somewhere between 5 and 30% of systems experience memory issues, yet high volume margin data collected at the system assembly factory suggests that only about 0.0064% of systems would be susceptible to experiencing problems due to bus margin.
- Only 0.16% of the systems that required a DIMM replacement showed low bus margin in the factory study.
- Bus margin degradation over time is unlikely given that the population of DIMM replacements includes a large variety of different DIMMs from 5 different DRAM manufacturers, eliminating any systemic problems.

It should be noted that this data was from a specific CPU family and set of product design requirements. The data suggests that the systems are well designed from a bus integrity point of view. It is possible that other products may have inferior bus designs, higher memory error rates, and higher proportion of those memory errors attributable to bus marginality. Also, further aging studies including contact corrosion and degradation are under investigation to better understand how bus margins may change over several years. However, for a well-designed system, the data shows that bus marginality is a very small contributor to overall memory subsystem health.

References

[1] B. Schroeder, E. Pinheiro, and W. Weber. "DRAM Errors in the Wild: A Large-Scale Field Study", *SIGMETRICS/Performance '09*, June 15-19, 2009, Seattle, WA, USA

[2] T. C. May and M. H. Woods, "Alpha-Particle-Induced Soft Errors in Dynamic Memories", IEEE Transactions on Electron Devices 26, No. 1, 2-9, 1979

[3] J. F. Ziegler and W. A. Lanford, "Effect of Cosmic Rays on Computer Memories", Science 206, No. 4420, 776-788, 1979

[4] J. F. Ziegler and W. A. Lanford, "The Effect of Sea Level Cosmic Rays on Electronic Devices", IEEE International Solid-State Circuits Conference, 1980

[5] S. Mukherjee. "Computer Glitches from Radiation: A Problem with Multiple Solutions", Microprocessor Report, May 19, 2008

[6] T. J. Dell, "System RAS implications of DRAM soft errors", IBM Journal of Research and Development, Vol. 52, No.3, May 2008

[7] X. Li, M. C. Huang, K. Shen, and L. Chu, "An Empirical Study of Memory Hardware Errors in A Server Farm", HotDep Workshop, 2007

[8] "Soft Errors in Electronic Memory – A White Paper", Tezzaron Semiconductor, January 5, 2004, Naperville, IL, USA

Software Testing in Critical Embedded Systems: a Systematic Review of Adherence to the DO-178B Standard

Jacson Rodrigues Barbosa*, Auri Marcelo Rizzo Vincenzi*
*Instituto de Informática
Universidade Federal de Goiás, UFG
Goiânia-GO, Brazil
E-mail: {jacsonbarbosa,auri}@inf.ufg.br

Márcio Eduardo Delamaro†, José Carlos Maldonado†
†Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo, USP
São Carlos-SP, Brazil
E-mail: {delamaro,jcmaldon}@icmc.usp.br

***Abstract*—Computing is becoming increasingly critical as far as embedded applications are concerned. Depending on the software, its malfunction may have consequences varying from serious financial problems to the loss of human lives. In view of this, this paper presents a systematic review that investigates the evolution of the work-related activity of embedded software critical tests in order to assess the level of compliance of the works found in relation to the DO-178B standard (*Software Considerations in Airborne Systems and Equipment Certification*). The ultimate goal of this research is the composition of existing works to define a test process that incorporates the quality and DO-178B requirements considering the different levels of criticality.**

***Keywords*-software testing; critical embedded system; DO-178B.**

I. Introduction

Embedded systems are often critical computational modules for monitoring and control used together with physical devices such as robots, autonomous vehicles and unmanned aircraft. Some systems impose restrictions with regard to security, performance, reliability and other factors, since failures in these systems may result in danger to human lives, environmental hazards or high financial losses.

By aiming to ensure quality levels which will reduce the chances of these tragic events, the *Radio Technical Commission for Aeronautics* (RTCA), together with the *European Organization for Civil Aviation Equipment* (EUROCAE) have created the DO-178B standard, which provides a set of guidelines for the development and certification of embedded software systems and applications, since these devices cannot be marketed by the industry without the latter's approval of this standard [1].

Because of this, the National Institute of Science and Technology Critical Embedded Systems (INCT-SEC) has recently been created to establish a network of collaboration and research in critical embedded systems (CES) [2]. The present work supports the goals of the INCT-SEC, investigating the evolution of research in software testing of critical embedded systems through a systematic review (SR) and assessing the level of compliance of such research with the DO-178B, in an attempt to identify a set of works which could be used together in a methodology for CES testing.

This paper is organized into four sections. Section II presents the main concepts related to the DO-178B standard. Section III describes the SR planning. Section IV shows the results obtained after conducting the review. Section V presents our conclusions on the topic and suggests future work to be carried out in the field.

II. Background: the DO-178B standard

The DO-178B standard defines the software's demand levels by considering the effects (failure condition) produced if the software behaves abnormally. Table I shows this relationship.

Table I
Software levels and failure condition (adapted from [3])

Software Level	*Failure Condition*
A	Catastrophic
B	Hazardous
C	Major
D	Minor
E	No effect

In DO-178B, the test on critical embedded systems has aims that complement the software verification process, showing that such software meets the relevant requirements and reveals a high degree of certainty that the defects which could lead to unacceptable failure conditions were removed [1].

To meet these goals in the software testing process, the standard defines a set of five requirements.

A. Normal range test cases

Normal range test cases show the software's ability to respond to inputs and normal conditions; for instance, an entire input variable should be executed by using valid equivalence classes and limit values.

B. Robustness test cases

Robustness test cases demonstrate the software's ability to respond to inputs and abnormal conditions; for instance, an input variable must be performed by using values of invalid equivalence classes.

C. Requirement-based testing methods

There are three testing methods based on requirements: integration of requirement-based hardware/software, integration testing of requirement-based software and low-level requirement-based test.

D. Requirement-based test coverage analysis

The purpose of this analysis is to determine how the implemented software requirements were verified with the requirement-based tests.

E. Structural coverage analysis

This analysis aims to show how the code structure was not executed by the requirement-based test.

Given the importance of the DO-178B standard as regards software certification for critical embedded systems used in aviation, one of the goals of INCT-SEC is to develop software for the control of unmanned aerial vehicles. The following section shows the planning of the systematic review conducted to identify previously developed research in this area of expertise.

III. Systematic review planning

The systematic review (SR) was planned following the model proposed by [4]. Figure 1 shows the SR's development process.

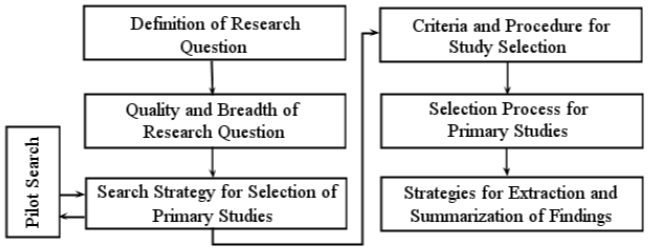

Definition of Research Question
Quality and Breadth of Research Question
Search Strategy for Selection of Primary Studies
Pilot Search
Criteria and Procedure for Study Selection
Selection Process for Primary Studies
Strategies for Extraction and Summarization of Findings

Figure 1. Systematic review process (adapted from [5])

A. Definition of Research Question

The purpose of the SR was to find answers to the following questions:

- **Primary Research Question 1 (PRQ1):** What techniques and software testing criteria have been proposed for software testing in critical embedded systems?
- **Secondary Research Question 1.1 (SRQ1.1):** What standards have been proposed for software testing in critical embedded systems?
- **Primary Research Question 2 (PRQ2):** What is the degree of adherence of experimental studies related to the objectives and activities of the software testing process defined in DO-178B?
- **Secondary Research Question 2.1 (SRQ2.1):** What evidence is there to confirm that the objectives and activities of the software testing process defined in DO-178B provide high quality standards in critical embedded systems?
- **Primary Research Question 3 (PRQ3):** What has been the strength of evidence supporting the conclusions drawn?

B. Quality and Breadth of Research Question

A well-formulated research question includes the following elements:

1) Keywords and Synonyms: the following were regarded as keywords in English:

- critical embedded, safety-critical, mission-critical, embedded software
- software test, system test

2) Intervention: software testing processes, techniques and criteria were observed in this review.

3) Control: we identified eight articles relevant to the context of this work , which served as control items of the search string. If the search string came up with all these articles, then that would confirm its appropriateness.

4) Population: the group was observed by researchers and software developers working on the design and construction of critical embedded systems.

5) Findings: software verification and validation (V&V) activities, software testing methodology, techniques and criteria for testing software used in the context of critical embedded systems.

6) Application: software development projects implemented within the context of critical embedded systems.

C. Search Strategy for Selection of Primary Studies

By taking into account the keywords, study sources, language and types of primary study, the following were selected for review:

1) Listing sources: electronic indexed databases IEEE Xplore (IEEE) and ACM Digital Library (ACM).

2) Language of primary studies: English.

3) Type of primary studies: reference lists of primary studies, journals, technical reports and conference proceedings.

D. Pilot Search

From the research questions and their respective attributes of quality and breadth, a search string was defined in order to perform an initial evaluation: *(critical embedded OR safety-critical OR mission-critical OR embedded software). AND (test) AND (software OR system)*

E. Criteria and Procedure for Selection of Studies

1) Inclusion criteria: the following inclusion criteria were defined:

- IC_1 – implementation of software V&V (static and/or dynamic) activities in the context of critical embedded systems;
- IC_2 – application of techniques and test criteria (dynamic) for software in critical embedded systems.

2) Exclusion criteria: the following exclusion criteria were defined:

- EC_1 – implementation of software V&V activities in the context of non-critical embedded systems e.g. mobile applications;
- EC_2 – application of techniques and criteria for software testing in non-critical embedded systems;
- EC_3 – does not address the activities of software V&V or techniques and criteria for software testing in the context of critical embedded systems.

F. Selection Process for Primary Studies

1) Primary Selection Process: search strings were formed by combining synonyms of the keywords identified. These strings were used to conduct searches in the search engines mentioned. The studies found through this research were analyzed by two reviewers (co-authors of this paper), who read and reviewed their titles and abstracts to rate them in terms of importance. If the reviewers reached an agreement over a given article, the manuscript was selected to be read in full.

2) Final Selection Process: we performed a thorough reading of papers selected in the preliminary stage by at least one of the reviewers.

3) Evaluation of Primary Studies: all primary studies were assessed individually by the reviewers based on the criteria defined in [5]. Reviewers then produced a document containing the summary, methodology and testing techniques mentioned in the primary studies, as well as other related concepts.

G. Strategies for Extraction and Summarization of Findings

For each primary study selected, we used the JabRef tool to store the collected data [6].

The summary of the results collected was organized into a tabular format.

IV. Data analysis

In Figure 2, Phase 1 amounts to the number of primary studies found by indexed electronic databases after submitting the query string (n=872). Phase 2 shows the number of studies resulting from the primary selection process (n=285); the remaining n=587 were excluded because their titles and summaries did not address the SR's scope of research questions. In Phase 3, n=185 were eliminated after the reading because they failed to meet the SR's full scope, thereby leaving n=100 primary studies. Finally, in Phase 4 n=3 were eliminated following the evaluation of primary studies according to quality criteria defined in the SR planning; they were considered of low quality. Thus, n=97 primary studies selected for extraction and summary of results remained. These phases were carried out by the authors during a period of five months. Further details about the primary studies selected are available in [7].

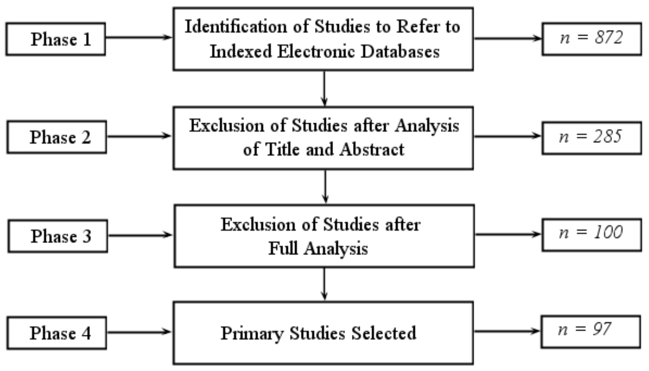

Phase 1
Identification of Studies to Refer to Indexed Electronic Databases
n = 872
Phase 2
Exclusion of Studies after Analysis of Title and Abstract
n = 285
Phase 3
Exclusion of Studies after Full Analysis
n = 100
Phase 4
Primary Studies Selected
n = 97

Figure 2. Phases of the final selection, adapted from [4]

Tables II, III and IV summarize the data of 97 primary studies and show partial quantities (IEEE and ACM), total studies (n) and total percentage of studies (%).

Table II presents quantitative information about the type of experimental study employed in the papers selected. This classification is based on the terminology defined by [4], according to which multiple-case refers to projects that include more than one case. By examining the table, it seems that 59.79% of the studies are observational (single-case and multiple-case), thus indicating that the majority are a result of monitoring one or more projects in depth.

Table II
Type of experimental study

Experimental Study	IEEE	ACM	n	%
Single-case	28	20	48	49.48
Multiple-case	4	6	10	10.31
Experiment	5	12	17	17.53
Survey	3	0	3	3.09
Not mentioned	12	7	19	19.59
Total	52	45	97	100

Table III shows the software testing techniques employed by the studies; if one approaches more than one technique, for instance quantity (q) equal to 3, a value of 0.33 (q^{-1}) would be assigned for each technique. It appears that the functional testing technique is most frequently used (36.77%), followed by model-based testing (28.36%), which allows us to eliminate ambiguities and to derive test cases from the model. The paper in [8] proposes the transition coverage criterion based on security requirements as a new alternative for the model-based testing technique.

Table III
Software testing techniques explored

Software testing	IEEE	ACM	n	%
Model-based testing	14.5	13	27.5	28.36
Mutation testing	0.33	1.83	2.16	2.23
Structural testing	8.83	14.83	23.66	24.4
Functional testing	21.33	14.33	35.66	36.77
Not mentioned	7	1	8	8.25
Total	51.99	44.99	96.98	100

Among the primary studies selected in the SR, the *Safety Critical Application Development Environment* (SCADE) has been widely quoted to specify critical embedded software, since the SCADE Suite allows the automatic generation of C code from specific models, such as state machines.

As regards the software testing criteria explored by the studies in question, the equivalence partition criterion was the most frequently used (12.37%). As far as structural testing criteria are concerned, the *Modified Condition/Decision Coverage* MC/DC was the most frequently used (10.23%). The remaining structural criteria required by DO-178B have been explored as follows: *Decision Coverage* DC (0.68%) and *Statement Coverage* SC (5.82%) , in response to PRQ1.

With respect to the norms and standards related to the development of critical embedded software (SRQ1.1), the DO-178B standard was the most frequently used (20.92%), thus indicating that the objectives and activities defined in DO-178B provide conditions to build a critical embedded software quality (in response to SRQ2.1). Furthermore, standards were used for specific industrial contexts; for instance, the standards set by the *European Committee for Electrotechnical Standardization* (CENELEC) are recommendations for the development and testing of rail transport systems [9].

As can be seen in Table IV, the requirement *Structural coverage analysis* of DO-178B has been extensively investigated - this is possibly due to the complexity associated with it. It has been observed that 36.09% of the studies carried out are not directly mappable to DO-178B test requirements, due to the fact that most studies address issues related to the definition of software life cycle models or other critical embedded software processes (responding to PRQ2). Further detailed information on this topic can be found in [7].

To meet software level *A*, the criteria for structural testing (MC/DC) must be adhered to, as these are the most rigorous structural criteria defined by DO-178B. The article in [10] presents a case study that meets the criteria when looking for MC/DC. Important errors that failed to be identified by functional technique were found by MC/DC, thus demonstrating its effectiveness for identifying critical bugs and for complementing the functional technique.

However, the study in [8] presents a subsumption hierarchy which compares the MC/DC and other criteria, thereby confirming that the *Multiple-Condition Coverage* test (M-CC) is more stringent than the MC/DC. In terms of overall effectiveness for fault detection, the following tests stand in decreasing order: MC/DC < CUTPNFP < MUMCUT < M-CC.

Table IV
Adherence to testing process requirements

DO-178B Testing Process Requirements	IEEE	ACM	n	%
Normal range test cases	4.61	2.91	7.52	7.75
Robustness test cases	4.48	2.58	7.06	7.28
Requirement-based testing methods	3.11	4.33	7.44	7.67
Requirement-based test coverage analysis	9.65	3.58	13.23	13.64
Structural coverage analysis	10.15	16.58	26.73	27.56
No direct mapping to DO-178B requirements	20	15	35	36.09
Total	52	44.98	96.98	100

By comparing Tables III and IV, it appears that the functional testing technique was the most frequently used, but the first two requirements of the DO-178B testing process shown in Table IV, which adhere to the functional test criteria (partitioning equivalence and boundary value analysis), have few related works. This is because the corresponding functional test criteria employed were not specified in this subset of primary studies.

As regards study characteristics, 59.79% of the studies selected are observational (as shown in Table II), whereas only 17.53% correspond to experiments. In accordance with the guidelines of the Grading of Recommendations Assessment, Development and Evaluation (GRADE), the evidence obtained by the SR concerning study characteristics are considered low (refer to [4], [5] for an overview).

Regarding the quality of the studies selected, their approaches to data analysis were explained in a moderate way, including issues such as potential bias, credibility and study limitations. Only in one study did the researcher critically assess his own role. Credibility was discussed in 98.45% of the studies, whereas study limitations were discussed in 72.68% of them. Based on the quality criterion results, the studies show moderate evidence.

As far as the consistency criterion is concerned, we identified similarities between 63.91% of the studies regarding DO-178B requirements (as shown in Table IV), since at least one of these requirements could be associated in more than a single primary study; the rest do not emphasize the requirements (36.09%). As a result, the strength of evidence regarding consistency may be classified as moderate.

Finally, we focused on the directness criterion, which assesses whether the people involved (students or software professionals), interventions and study results are consistent with the area of interest. In the SR, most studies were carried out in an academic context, and only one study mentioned the level of knowledge and experience of the students involved - both undergraduate and experienced graduate students. As regards intervention, objective comparisons were carried out with 63.91% of the studies concerning DO-178B requirements, even though only 20.92% of the studies clearly mentioned the use of the standard in question. Finally, as for the results obtained, even though most of the studies are observational, they are not trivial, being thus comparable with the software/systems developed by the industry. This information allows us to regard the strength of evidence related to objectivity as low to moderate.

Once the four criteria (characteristics, quality, consistency and directness) are combined to determine the strength of evidence for this RS , the latter may be classified as moderate, hence responding to PRQ3. Therefore, defining the strength of evidence may help future research to have a crucial impact on the reliability of effect estimates, hence changing current estimates [5].

V. Final considerations and future work

In the analysis of the studies selected, we found that all DO-178B testing process requirements have been explored, but few studies have discussed how to solve problems of structural coverage analysis (n=2), such as dead code and deactivated code.

In the future, we intend to propose a software test methodology that supports the INCT-SEC projects compliant with DO-178B and uses the activities of software verification and validation as well as the techniques and criteria for software testing identified in the systematic review. Since DO-178B does not define how to implement the respective processes, the methodology should state how the processes are to be implemented in accordance with the necessary requirement levels.

Moreover, it is possible to reuse the SR protocol further, collecting more data aiming at identifying how the state of the art evolved and the still missing pitfalls in the context of testing of critical embedded system (CES).

Acknowledgments

We are indebted to CNPq (573963/2008-8), FAPESP (08/57870-9), FAPEG (200910267000662) and CAPES for their financial support. We also thank professor Plínio de Sá Leitão Júnior for reviewing this paper.

References

[1] *Software considerations in airbone systems and equipament certification*, RTCA SC-167/EUROCAE WG-12, 1992.

[2] J. C. Maldonado, "National institute of science and technology critical embedded systems (INCT-SEC)," *São Carlos/SP - Brazil*, 2008. [Online]. Available: http://www.inct-sec.org/?q=en-us. Accessed on: [08/18/2011].

[3] T. K. Ferrel and U. D. Ferrel, *The Avionics Handbook*. CRC Press LLC, 2001.

[4] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and Software Technology*, 2008.

[5] M. S. Ali, M. Ali Babar, L. Chen, and K.-J. Stol, "A systematic review of comparative evidence of aspect-oriented programming," *Inf. Softw. Technol.*, vol. 52, pp. 871–887, September 2010.

[6] *JabRef 2.4*, 2008. [Online]. Available: http://jabref.sourceforce.net/. Accessed on: [08/18/2011].

[7] J. R. Barbosa and A. M. R. Vincenzi, "Software testing in the context of critical embedded systems," Web page, july 2011. [Online]. Available: http://www.inf.ufg.br/˜auri/sec-en/. Accessed on: [08/18/2011].

[8] Y. T. Yu and M. F. Lau, "A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions," *J. Syst. Softw.*, vol. 79, pp. 577–590, May 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2005.05.030. Accessed on: [08/18/2011].

[9] J. Kloos and R. Eschbach, "Generating system models for a highly configurable train control system using a domain-specific language: A case study," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 39–47.

[10] A. Dupuy and N. Leveson, "An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software," in *Digital Avionics Systems Conferences, 2000. Proceedings. DASC. The 19th*, vol. 1, 2000, pp. 1B6/1 –1B6/7 vol.1.