VALID 2013

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

ISBN: 978-1-61208-307-0

October 27 - November 1, 2013

Venice, Italy

VALID 2013 Editors

Jos van Rooyen, Bartosz, the Netherlands

Philipp Helle, EADS Innovation Works, Germany

Pascal Lorenz, University on Haute Alsace, France

VALID 2013

Forward

The Fifth International Conference on Advances in System Testing and Validation Lifecycle (VALID 2013), held on October 27 - November 1, 2013 - Venice, Italy, 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 2013 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 VALID 2013. 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 2013 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 2013 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 also hope the attendees enjoyed the charm of Venice.

VALID 2013 Chairs

VALID 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
Amir Alimohammad, San Diego State University, USA
Hema Srikanth, IBM, USA

VALID 2013 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

VALID 2013 Industry Chairs

Abel Marrero, Bombardier Transportation Germany GmbH - Mannheim, Germany
Sebastian Wieczorek, SAP AG - Darmstadt, Germany
Eric Verhulst, Altreonic, Belgium

VALID 2013

Committee

VALID 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
Amir Alimohammad, San Diego State University, USA
Hema Srikanth, IBM, USA

VALID 2013 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

VALID 2013 Industry Chairs

Abel Marrero, Bombardier Transportation Germany GmbH - Mannheim, Germany
Sebastian Wieczorek, SAP AG - Darmstadt, Germany
Eric Verhulst, Altreonic, Belgium

VALID 2013 Technical Progam Committee

Fredrik Abbors, Åbo Akademi University, Finland
Jaume Abella, Barcelona Supercomputing Center (BSC-CNS), Spain
Mehmet Aksit, University of Twente - Enschede, The Netherlands
Amir Alimohammad, San Diego State University, USA
Giner Alor Hernandez, Instituto Tecnologico de Orizaba - Veracruz, México
Nina Amla, NSF, USA
César Andrés Sanchez, Universidad Complutense de Madrid, Spain
Selma Azaiz, CEA List Institute - Gif-Sur-Yvette, France
Cesare Bartolini, ISTI - CNR, Pisa, Italy
Andrea Baruzzo, Università degli Studi di Udine, Italy
Serge Bernard, LIRMM, Franmce

Paolo Bernardi, Politecnico di Torino, Italy
Ateet Bhalla, Oriental Institute of Science and Technology, India
Bruno Blaškovic, Faculty of Electrical Engineering and Computing ZOEEM - CRS lab, Croatia
Mikey Browne, IBM, USA
Mark Burgin, University of California Los Angeles (UCLA), USA
Isabel Cafezeiro, Instituto de Computação - Universidade Federal Fluminense, Brazil
Luca Cassano, University of Pisa, Italy
Fouad Chedid, Notre Dame University, Lebanon
Hana Chockler, IBM Haifa Research Labs, Israel
Bruce F. Cockburn, University of Alberta - Edmonton, Canada
Maurizio M D'Arienzo, Seconda Università di Napoli, Italy
Florian Deissenboeck, CQSE GmbH/ Technische Universität München, Germany
Gülşen Demiröz, Sabanci University, Turkey
Stefano Di Carlo, Politecnico di Torino, Italy
Rolf Drechsler, DFKI Bremen, Germany
Lydie du Bousquet, J. Fourier-Grenoble I University / LIG labs, France
Kerstin Eder, University of Bristol, UK
Stephan Eggersglüß, University of Bremen / DFKI - Cyper-Physical Systems - Bremen, Germany
Khaled El-Fakih, American University of Sharjah, UAE
Leire Etxeberria Elorza, Mondragon Unibertsitatea, Spain
Eitan Farchi, IBM Haifa Research Laboratory, Israel
Michael Felderer, University of Innsbruck, Austria
Teodor Ghetiu, University of York, UK
Shalini Ghosh, Computer Science Laboratory - SRI, USA
Patrick Girard, LIRMM, France
Hans-Gerhard Gross, Delft University of Technology, The Netherlands
Bidyut Gupta, Southern Illinois University, USA
Mark Harman, University College London, UK
Kazumi Hatayama, Nara Institute of Science and Technology (NAIST), Japan
Steffen Herbold, University of Göttingen, Germany
Florentin Ipate, University of Pitesti, Romania
David Kaeli, Northeastern University - Boston, USA
Ahmed Kamel, Concordia College, USA
Teemu Kanstrén, VTT Technical Research Centre of Finland, Finland
Zurab Khasidashvili, Intel Israel Ltd, Israel
Alexander Klaus, Fraunhofer Institute for Experimental Software Engineering - Kaiserslautern, Germany
Weiqiang Kong, Kyushu University, Japan
Moshe Levinger, IBM Research - Haifa, Israel
João Lourenço, Universidade Nova de Lisboa, Portugal
Maria K. Michael, University of Cyprus, Cyprus
Abel Marrero, Bombardier Transportation Germany GmbH - Mannheim, Germany
Roy Oberhauser, Aalen University, Germany
Johannes Oetsch, Vienna University of Technology, Austria

Nguena-Timo Omer-Landry, LaBRI/University Bordeaux 1, France
Yassine Ouhammou, ENSMA / LIAS-lab, France
Kai Pan, University of North Carolina at Charlotte, USA
Bernhard Peischl, Softnet Austria, Austria
Juho Perälä, VTT Technical Research Centre of Finland, Finland
Mauro Pezzè, Università della Svizzera Italiana, Switzerland
Miodrag Potkonjak, Univeristy of California, Los Angeles (UCLA), USA
Wishnu Prasetya, Utrecht University, The Netherlands
Paolo Prinetto, Politecnico di Torino, Italy
Henrique Rebêlo, Federal University of Pernambuco, Brazil
Eike Reetz, University of Applied Sciences Osnabrück, Germany
Filippo Ricca, University of Genoa, Italy
Auri Marcelo RizzoVicenzi, Universidade Federal de Goiás, Brazil
Goiuria Sagardui Mendieta, Mondragon University, Spain
Christian Schanes, Vienna University of Technology, Austria
Cristina Seceleanu, Mälardalen University, Sweden
Nassim Seghir, University of Oxford, UK
Sergio Segura, University of Seville, Spain
Hema Srikanth, IBM, USA
Alin Stefanescu, University of Pitesti, Romania
Mehdi B. Tahoori, Karlsruhe Institute of Technology (KIT), Germany
Nur A. Touba, University of Texas - Austin, USA
Spyros Tragoudas, Southern Illinois University Carbondale, USA
Dragos Truscan, Åbo Akademi University - Turku, Finland
Jos van Rooyen, Bartosz ICT, Netherlands
Bart Vermeulen, NXP Semiconductors - Eindhoven, The Netherlands
Arnaud Virazel, Université Montpellier 2 / LIRMM, France
Tanja E. J. Vos, Universidad Politécnica de Valencia, Spain
Stefan Wagner, University of Stuttgart, Germany
Sebastian Wieczorek, SAP Research Center Darmstadt, Germany
Lina Ye, Inria Grenoble, France
Cemal Yilmaz, Sabanci University - Istanbul, Turkey
Zeljko Zilic, McGill University, Canada

Copyright Information

For your reference, this is the text governing the copyright release for material published by IARIA.

The copyright release is a transfer of publication rights, which allows IARIA and its partners to drive the dissemination of the published material. This allows IARIA to give articles increased visibility via distribution, inclusion in libraries, and arrangements for submission to indexes.

I, the undersigned, declare that the article is original, and that I represent the authors of this article in the copyright release matters. If this work has been done as work-for-hire, I have obtained all necessary clearances to execute a copyright release. I hereby irrevocably transfer exclusive copyright for this material to IARIA. I give IARIA permission or reproduce the work in any media format such as, but not limited to, print, digital, or electronic. I give IARIA permission to distribute the materials without restriction to any institutions or individuals. I give IARIA permission to submit the work for inclusion in article repositories as IARIA sees fit.

I, the undersigned, declare that to the best of my knowledge, the article is does not contain libelous or otherwise unlawful contents or invading the right of privacy or infringing on a proprietary right.

Following the copyright release, any circulated version of the article must bear the copyright notice and any header and footer information that IARIA applies to the published article.

IARIA grants royalty-free permission to the authors to disseminate the work, under the above provisions, for any academic, commercial, or industrial use. IARIA grants royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

IARIA acknowledges that rights to any algorithm, process, procedure, apparatus, or articles of manufacture remain with the authors and their employers.

I, the undersigned, understand that IARIA will not be liable, in contract, tort (including, without limitation, negligence), pre-contract or other representations (other than fraudulent misrepresentations) or otherwise in connection with the publication of my work.

Exception to the above is made for work-for-hire performed while employed by the government. In that case, copyright to the material remains with the said government. The rightful owners (authors and government entity) grant unlimited and unrestricted permission to IARIA, IARIA's contractors, and IARIA's partners to further distribute the work.

Table of Contents

Model-Based MCDC Testing of Complex Decisions for the Java Card Applet Firewall — 1
Roderick Bloem, Karin Greimel, Robert Koenighofer, and Franz Roeck

Enabling Interface Validation through Text Generation — 7
Hakan Burden, Rogardt Heldal, and Peter Ljunglof

Efficient Elimination of False Positives Using Bounded Model Checking — 13
Tukaram Muske, Advaita Datar, Mayur Khanzode, and Kumar Madhukar

State Space Reconstruction for On-Line Model Checking with UPPAAL — 21
Jonas Rinast, Sibylle Schupp, and Dieter Gollmann

Formal Composition based on Roles within a Model Driven Engineering Approach — 27
Cedrick Lelionnais, Jerome Delatour, Matthias Brun, Olivier H. Roux, and Charlotte Seidner

Preliminary Test Suite Reduction — 33
Vitaly Kozyura and Sebastian Wieczorek

Performance Characterization of TAS-MRAM Architectures in Presence of Capacitive Defects — 39
Joao Azevedo, Arnaud Virazel, Yuanqing Cheng, Alberto Bosio, Luigi Dilillo, Patrick Girard, Aida Todri, and Jeremy Alvarez Herault

Automatic Linking of Test Cases and Requirements — 45
Thomas Noack

Using Filtering to Improve Value-Level Debugging of Verilog Designs — 49
Bernhard Peischl, Naveed Riaz, and Franz Wotawa

Towards an Integrated Methodology for the Development and Testing of Complex Systems — 55
Philipp Helle and Wladimir Schamai

An Evaluation of Client-Side Dependencies of Search Engines by Load Testing — 61
Emine Sefer and Sinem Aykanat

Compact Traceable Logging — 66
Wishnu Prasetya, Ales Sturala, Arie Middelkoop, Jurriaan Hage, and Alexander Elyasov

Model-Based MCDC Testing of Complex Decisions for the Java Card Applet Firewall

Roderick Bloem[1], Karin Greimel[2], Robert Koenighofer[1], Franz Roeck[1,2]

[1]Institute for Applied Information Processing and Communications
Graz University of Technology, A-8010 Graz, Austria
{roderick.bloem, robert.koenighofer, franz.roeck}@iaik.tugraz.at
[2]NXP Semiconductors Austria GmbH, Gratkorn, A-8101 Gratkorn, Austria
{karin.greimel, franz.roeck}@nxp.com

***Abstract*—Certification processes require the generation of models of a design. Using Model-Based Testing, these models can double as guides for test case generation. In this paper, we consider Boolean formulas that model a decision to be taken by a part of the software. We show how to use an SMT-solver to generate test cases that fulfill the MCDC coverage criteria on these models, in the presence of *strong coupling*. We show that the approach can improve test coverage, and finds a bug in an implementation of the Java Card Applet Firewall.**

Keywords*—automatic test case generation; common criteria; java card applet firewall.

I. Introduction

Certification of security critical embedded systems at a certain level requires that formal models of the design are created and verified against the security requirements. The main motivation for the work presented in this paper is to complement this certification effort with systematic testing. Re-using existing models for test case generation, we can increase the confidence in the quality of the actual implementation at little extra cost.

Common Criteria [5] is a typical, widely used certification scheme. It assures that *Security Functional Requirements* are met by the *Target of Evaluation*. It offers several *Evaluation Assurance Levels* (EAL). Starting with EAL6, a formal model is required to prove that the Security Functional Requirements are satisfied. For complexity reasons, this proof is (typically) carried out on the model and *not* on the actual implementation. We propose to complement the certification with test cases derived automatically from the model in order to close the link from the security functional requirements down to the actual implementation, as illustrated in Fig. 1. The arrow from the model to the implementation is dashed to emphasize that the implementation is often not derived from the model but developed independently. Thus, it is important to perform a conformance check, and test cases are a scalable and flexible option.

Models of security-critical systems often contain complex decisions, i.e., expressions evaluating to true or false. They may express, for instance, under which circumstances a user login should be successful or access to some resource should be allowed. Complex decisions may directly serve as models for stateless parts of the system (e.g., a method that checks if some access is allowed). They may also appear as guards in transition systems modeling stateful parts of the design. Such complex decisions are often difficult to test manually. First, there may be complex dependencies between the different parts of the decisions. Second, exhaustive testing is often infeasible, but we still want to cover the "interesting" cases. Test criteria help by defining which cases are interesting and have to be tested and we can use them to automatically generate test cases. The benefit of this Model-Based Testing approach is that the models are much simpler than the implementation, but precisely describe the various cases of interest. Also, the model acts as a test oracle. In our setting, the model is derived as a side-effect of the certification procedure, at no extra cost.

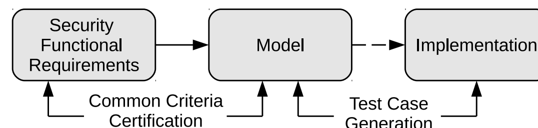


Fig. 1: Test case generation complements certification.

One widely used test criterion to measure code coverage is the Modified Condition Decision Coverage (MCDC) [9]. It is required by the US Federal Aviation Administration for safety critical software in aircrafts [13], and also used in many other domains. While MCDC is mostly used to measure the coverage of test cases with respect to the decisions in the implementation, we will apply the criterion to generate test cases from the decisions in the model.

In this paper, we show how to derive a test suite that achieves MCDC on a model that consists of logical decisions. Using a Satisfiability Modulo Theories (SMT) solver, we obtain values for the variables used in the decision. That is, our method not only computes the desired truth values of the different conditions (i.e., atoms) in a decision, but it also computes values for the (potentially non-Boolean) variables, which can then be used as test cases. It can handle complex interdependencies ("couplings") between the conditions by passing them on to the solver.

We apply our implementation in a case study of a Java Card applet firewall. This firewall is modeled as a decision under which access to an object is granted. Although a large set of manually constructed tests exists, the automatically constructed tests increase the code coverage. Using the model as a test oracle, our test suite also detects an inconsistency between the implementation and the specification due to an update of the reference manual which was not implemented.

This work was supported in part by the Austrian Research Promotion Agency (FWF) through project NewP@ss (835917).

Our test case generation method can be applied directly to testing Boolean effects in stateless systems. Our approach can also be combined with that of [19] to obtain test cases for systems that are modeled as transition systems, obtaining a test suite that exercises the guards of the state transitions.

Several papers on formal modeling for high assurance Common Criteria evaluations exist [4], [18], [7]. Both in [18] and in [7], the Java Card firewall is modeled and a theorem prover is used to prove that the model satisfies the access control policy. For a Common Criteria certification it is not necessary to formally link the model to the implementation. In [7], the gap to the implementation is closed by manual code-to-spec review. In contrast, we propose to close this gap by generating test cases from the formal model and run them on the implementation. In [16], the functional correctness of an OS kernel is directly proven for the implementation. This gives higher assurance of the correctness but increases the effort tremendously. Our approach of integrating formal verification at a high abstraction level and model-based testing at the implementation level gives a good trade-off between assurance and cost.

The idea of generating test cases based on formal specifications was already presented in [3]. Since then, a lot of research has been done in this field [21], [19], [14], [22]. In [14] a survey on testing with model checkers is given. Using model checkers, test cases are generated by defining trap properties in CTL formulas such that a counterexample represents a test case. The disadvantage of this approach, however, is that model checking can be quite resource intensive. Feeding the guards of the transitions into an SMT-solver, as we do, is potentially cheaper. The closest related work we are aware of regarding test case generation is presented in [19]. The authors compute test cases achieving MCDC on a specification by walking through the parse trees of the decisions. Depending on the logical operator they decide what the expression of the subtree should evaluate to. In contrast, our method (a) does not stop at the Boolean level but also produces values for non-Boolean variables appearing in the decisions, and (b) can handle complex dependencies between the different parts of the decision.

The rest of this paper is structured as follows. Section II introduces background and notation, and gives an example. Section III presents our quality assurance flow based on certification and test case generation. Section IV discusses our case study with the Java Card applet firewall, and Section V draws conclusions and gives ideas for future work.

II. PRELIMINARIES

A. Decisions and Specifications

Let V be a set of variables ranging over a domain $\mathbb{D}$, and let F be a set of function symbols. A *term* over V and F is defined inductively as follows: (a) any variable $v \in V$ is a term, and (b) if $f \in F$ is a function symbol with arity n and $a_1, a_2, \dots a_n$ are terms, then $f(a_1, a_2, \dots a_n)$ is a term. For simplicity of the presentation, we assume that all variables have the same domain $\mathbb{D}$. E.g., $\mathbb{D}$ could be the domain of integers or bit-vectors of length 32. Also, all functions $f \in F$ are mappings $f : \mathbb{D} \times \dots \times \mathbb{D} \to \mathbb{D}$. A *condition* is a function mapping a vector of terms to either true ($\top$) or false ($\bot$). A *decision* φ is defined inductively as follows: (a) every condition is a decision, and (b) if φ_1 and φ_2 are decisions, then $\neg\varphi_1$ and $\varphi_1 \vee \varphi_2$ are decisions as well. The Boolean operators $\neg$ and $\vee$ have their usual semantics. Other Boolean operators can be seen as shortcuts. A *specification* is a set $S = \{\varphi_1, \varphi_2, \dots\}$ of decisions.

We write $\mathsf{CoN}(\varphi) = \{c_1, c_2, \dots\}$ for the set of all condition nodes in the parse tree of decision φ, and $\varphi[c|\top]$ ($\varphi[c|\bot]$) for the decision φ with condition node $c \in \mathsf{CoN}(\varphi)$ replaced by $\top$ ($\bot$).

B. Test Cases and Specification Coverage

A *test case* for a specification $S = \{\varphi_1, \varphi_2, \dots\}$ is an assignment $t : V \to \mathbb{D}$ of values to all variables in V. We write $\varphi(t)$ or $c(t)$ to denote the truth value ($\top$ or $\bot$) of decision φ or condition node $c \in \mathsf{CoN}(\varphi)$ under assignment t. A *test suite* is a set $T = \{t_1, t_2, \dots\}$ of test cases.

Let φ be a decision, $c \in \mathsf{CoN}(\varphi)$ be a condition node, and $t : V \to \mathbb{D}$ be a test case. We say that c *determines* φ under t, written $\mathsf{det}(c, \varphi, t)$, iff $\varphi(t) \neq \varphi[c|\neg c(t)](t)$. That is, negating the truth value of c changes the truth value of φ.

Test suite T achieves *Masking Modified Condition Decision Coverage* [8] on specification S iff for all $\varphi \in S$:

$$\exists t, t' \in T : \varphi(t) \wedge \neg\varphi(t') \quad (1)$$

and

$$\begin{aligned} &\forall c \in \mathsf{CoN}(\varphi) : \exists t, t' \in T : \\ &\quad c(t) \wedge \neg c(t') \wedge \mathsf{det}(c, \varphi, t) \wedge \mathsf{det}(c, \varphi, t'). \end{aligned} \quad (2)$$

That is, every decision $\varphi \in S$ must evaluate to true and to false on some test. Also, every condition node c must evaluate to true and to false while determining the truth value of φ. Masking MCDC is also referred to as *Correlated Active Clause Coverage* in the literature [1].

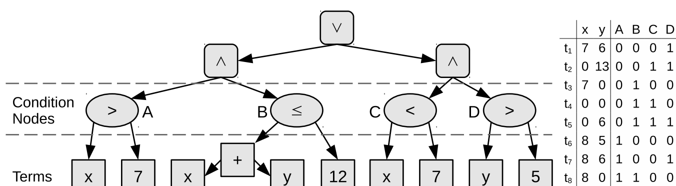
Unique cause MCDC [8] is a stricter variant. Whereas masking MCDC allows other occurring conditions to evaluate to different truth values for t and t' as long as the determination of c is preserved, unique cause MCDC requires the conditions to be same for both t and t'. Expressed more formally, we say that test suite T achieves unique cause MCDC on specification S iff for all $\varphi \in S$:

$$\exists t, t' \in T : \varphi(t) \wedge \neg\varphi(t') \quad (3)$$

and

$$\begin{aligned} &\forall c \in \mathsf{CoN}(\varphi) : \exists t, t' \in T : \\ &\quad c(t) \wedge \neg c(t') \wedge \mathsf{det}(c, \varphi, t) \wedge \mathsf{det}(c, \varphi, t') \wedge \\ &\quad \forall c' \in \{\mathsf{CoN}(\varphi) \backslash c\} : c'(t) = c'(t'). \end{aligned} \quad (4)$$

MCDC (either kind) can be calculated straightforward as long as all conditions are independent. However, variables may occur in more than one condition, and fixing the truth value of some conditions may determine others. If the truth value of one condition always flips when flipping the truth value of another condition, then these conditions are called *strongly coupled* [9]. If it flips in some but not in all cases, they are called *weakly coupled*. E.g., $(A > 5)$ and $(A < 9)$ are weakly coupled, whereas $(A > 5)$ and $(A \leq 5)$ are strongly



	x	y	A	B	C	D
t_1	7	6	0	0	0	1
t_2	0	13	0	0	1	1
t_3	7	0	0	1	0	0
t_4	0	0	0	1	1	0
t_5	0	6	0	1	1	1
t_6	8	5	1	0	0	0
t_7	8	6	1	0	0	1
t_8	8	0	1	1	0	0

Fig. 2: Example: Parse tree of a decision with test cases.

coupled. Due to coupled conditions, some test cases required for achieving MCDC may be infeasible.

Another metric, closer to exhaustive testing, is *Multiple Condition Coverage (MCC)*. MCC requires that for every decision, all possible combinations of the truth values of its conditions are tested. This results in 2^n test cases for n conditions in a single decision.

C. Example

Fig. 2 depicts the parse tree of the decision $\varphi = (x > 7 \wedge x + y \leq 12) \vee (x < 7 \wedge y > 5)$. Even though the decision is small, it is certainly not easy to test due to couplings between conditions. The parse tree contains four condition nodes, but from the $2^4 = 16$ possible truth value combinations, only 8 are satisfiable. They are listed as potential test cases t_1 to t_8 at the right side of the figure.

The four test cases t_2, t_4, t_7, t_8 achieve masking MCDC as follows. Condition node A is tested by t_4 and t_8: $\varphi(t_4) = \bot \neq \varphi(t_4)[A|\top] = \top$, so A determines φ under t_4. Analogously for t_8. In the same way, B is tested by t_7 and t_8, C by t_2 and t_7, and D by t_2 and t_4. However, these four test cases do not achieve unique cause MCDC because the pairs of test cases do not only flip the truth value of the tested condition node, but also others.

Unique cause MCDC can be achieved by the seven test cases $t_1, t_2, t_3, t_4, t_5, t_6, t_8$. Condition node A is now tested by t_3 and t_8. We have that $B(t_3) = B(t_8) = \top$, $C(t_3) = C(t_8) = \bot$, and $D(t_3) = D(t_8) = \bot$, so the truth value for the other condition nodes remains the same when testing A. In a similar way, condition node B is tested by t_6 and t_8, C by t_1 and t_2, and D by t_4 and t_5.

III. Certification with Test Case Generation

This section presents our proposed quality assurance flow, which is based on certification and automatic test case generation. It consists of the following four steps (see also Fig. 1):

1) Construct a model of the system.
2) Prove that the model of the system satisfies the requirements.
3) From the provenly correct model, automatically generate test cases.
4) Run the tests on the implementation.

Obviously, this flow does not give a formal proof of correctness for the implementation. It only proves that the model is correct with respect to the requirements. The test cases then verify that the model has been implemented correctly.

A. Certification

For certification under high assurance levels, like Common Criteria EAL6 or EAL7, formal models of the specification and of the design are required at different levels of detail, depending on the level of the certification. These models describe how the product implements certain parts of the specification. For Common Criteria, the requirements for formal security policy models are given in [6]. The 'Bundesamt für Sicherheit in der Informationstechnik', one of the certification bodies, published a guideline for the evaluation of security policy models [17]. The guideline suggests to use formal tools such as theorem provers or model checkers. We use model checking because these tools work fully automatically. A model checker takes a formal specification and a model as input. It returns `true` if the model satisfies the specification, giving a mathematical proof of correctness. Otherwise, if the model does not satisfy the specification, the model checker returns a counterexample [11].

We use the model checker NuSMV [10] for certification, as described in [4] for a smart card system. NuSMV has a proprietary modeling language, defining a finite state machine. A model consists of state and input variables, and of transitions defining how an input leads from one state to the next states. The rules on the transitions can be complex logical decisions. In the next section we will explain how to automatically generate test cases from complex logical decisions with high coverage.

B. Test Case Generation

We created a tool to compute a test suite T that achieves MCDC on a specification S. It uses the SMT-solver Z3 [12] to compute test cases as satisfying assignments of the constraints that have to be fulfilled by the tests. The decisions of the specification must be given in SMT-LIB2 format [2]. Our tool builds a parse tree of the decisions so that condition nodes can be found and replaced by $\top$ or $\bot$ easily. Next, it passes the constraints of Eq. 2 to the solver, one after the other, for all $\varphi \in S$ and $c \in \mathsf{CoN}(\varphi)$. For each satisfiable query, we can extract a pair of test cases t, t' as a satisfying assignment and add it to the test suite T. Variables which are irrelevant for the satisfying assignment will get a value which is either random or defined by the user. Unsatisfiable cases are reported to the user, because they usually indicate inconsistencies or redundancies in the decision. Before adding a new pair of test cases, we check if T already contains test cases that satisfy the constraints. This reduces the overall number of test cases. Finally, the test suite T is written into a simple text file, which can be parsed by a test adapter. Our tool supports masking MCDC, unique cause MCDC, and MCC.

IV. Case Study: Java Card Firewall

We applied our test case generation tool to the guard expressing the access rules in the formal model of the Java Card applet firewall, as specified in Section 6.2.8 of the JCRE specification, version 3 [20]. As a result, we obtain a set of test cases satisfying the MCDC criterion with respect to the guard in the model. Using MCDC ensures that each sub-item of the specification independently affects the access decision. After running the test suite it can then be assured, provided that none of the SMT-solver calls were unsatisfiable, that every

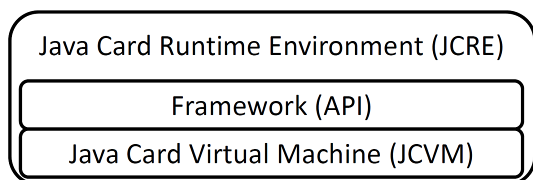


Fig. 3: The Java Card Runtime Environment.

explicit requirement from the specification is implemented and evaluates, for the state which is used for testing, to the expected outcome. The generation of the test suite for the guard, consisting of 223 conditions, takes less than a minute and results in 205 test cases before removing duplicates and 127 test cases after the elimination. As some variables like the one to identify the current bytecode, the current context, the object owner and so on, occur in more than one condition, some SMT-solver calls were unsatisfiable due to coupled conditions.

A. *Java Card Applet Firewall Model*

Whereas in standard Java every applet runs on its own instance of a virtual machine, the Java Card virtual machine must be able to deal with several (independent) applets. The Java Card applet firewall ensures that applets cannot randomly access data belonging to other applets, but only in restricted cases. The applet firewall is part of the Java Card virtual machine (JCVM) (see Fig. 3) and checks every single access according to the JCRE specification [20].

Our model of the Java Card firewall (see Fig. 4) consists of only two states: `idle` and `locked`. As long as the firewall is in the `idle` state, it performs the required checks for the Java Card virtual machine. If an access is denied, a *SecurityException* is thrown. The JCVM then has to handle this exception, e.g. reset a started transaction, while the applet firewall doesn't do anything. This situation we have modeled by introducing the second state called `locked`. Conceptually, this behavior is very simple. The difficulty for testing the firewall stems from the very complicated decision when to allow access.

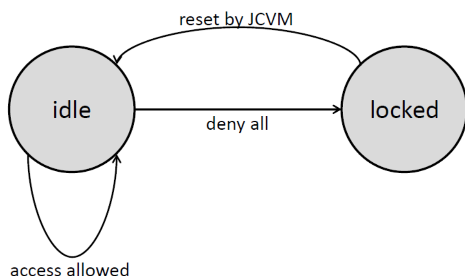


Fig. 4: Abstract model of the Java Card applet firewall.

The access rules are modeled by two transitions. The first transition, representing access allowed, is a self loop of the idle state. The second transition is a *deny all* transition, going from idle to locked with a lower priority. This realization ensures that every access, which is not explicitly allowed, is denied. The guard of the self loop is a formalization of Section 6.2.8 of the JCRE specification [20] such that a satisfying assignment for the formula corresponds to an access which is allowed.

Example 1: Section 6.2.8.7 of the JCRE specification [20] specifies access rules for the bytecode `athrow` by saying:

- "If the object is owned by an applet in the currently active context, access is allowed.
- Otherwise, if the object is designated a Java Card RE Entry Point Object, access is allowed.
- Otherwise, if the Java Card RE is the currently active context, access is allowed.
- Otherwise, access is denied."

This can be translated into a formula

$$\begin{aligned}&(\text{bytecode} = 7)\wedge\\&((\text{Owner} = \text{FLAG_CurrentlyActiveContext})\vee\\&(\text{FLAG_entryPointJCREObject})\vee\\&(\text{FLAG_CurrentlyActiveContext} = 0)),\end{aligned} \tag{5}$$

where $(\text{bytecode} = 7)$ checks if the bytecode equals *athrow*, and the remaining three lines correspond to the first three bullets copied from the JCRE specification, with the constant 0 encoding the JCRE context. This example illustrates that formulating the specification of the firewall is (for the most part) rather straightforward.

B. *Evaluation Setting*

We compare the quality of the test cases created automatically using our tool to that of the JCTCK, a hand-crafted test suite. The hand-crafted tests are given as Java Card applets. They test the whole implementation of the Java Card runtime environment and not only parts of it. The test harness for these tests simply runs the applets. In contrast to that, our test adapter runs the test cases as module tests implemented in C, which is the language the Java Card operating system is programmed in. It sets up the memory as required from the test case and calls the relevant bytecode implementation which performs the necessary firewall checks. If access is denied, a security exception is thrown, otherwise access was allowed.

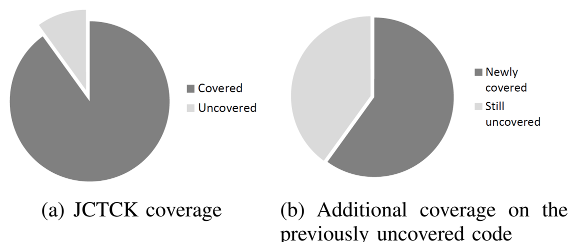
To evaluate the code coverage for both test suites, the functions belonging to the Java Card applet firewall were instrumented using a code coverage tool. The code was instrumented in a way such that covering all instrumentations corresponds to condition coverage plus basic block coverage. After running the test suites, analysis can be performed on the collected data and the code coverage can be compared.

Of course, the test cases do not only have the purpose of covering as much code as possible, but also to check if the applet firewall's behavior conforms to the JCRE specification. The provided JCTCK installs and runs the applets and the applets themselves check if the result corresponds to the expected outcome. The test adapter for our automatically constructed test cases calculates the expected result, namely either access allowed or denied, by evaluating the decision in the model on the test case input data. If the access is denied, a security exception is thrown by the implementation

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

TABLE I: Instrumentations and achieved coverage

test suite	covered / total	percentage
JCTCK	64/71	90%
our test suite	63/71	89%
together	68/71	96%



(a) JCTCK coverage (b) Additional coverage on the previously uncovered code

Fig. 5: Additional coverage on previously from the JCTCK uncovered code achieved by our test suite.

and caught by the test adapter. The test adapter finally checks for discrepancies between occurred and expected exceptions.

Our test approach is quite different from that of the JCTCK, so we are going to evaluate only results which are targeted from both test adapters. Moreover, we will explicitly mention if certain test goals can not be reached due to limitations stemming from the type of test. One such limitation is, for example, that a Java Card applet is restricted in its object creation.

C. Code Coverage Results

As we consider Section 6.2.8 of the JCRE specification as base for our test suite, we only evaluate the coverage in the functions of the applet firewall dealing with this Chapter. In some of the functions, the end can not be reached due to thrown exceptions. The total number of 78 instrumentations for the coverage is reduced by those, such that a coverage of all remaining 71 instrumentations corresponds to 100 % coverage. After running the test suites and storing the results for each test suite, the code coverage investigations reveal that neither of the test suites did achieve a full condition plus basic block coverage of 100 % (see Table I). However, when using both test suites it is possible to increase the coverage of the JCTCK by six percent, from 90 % to 96 %. This means that the automatically generated test suite covers 60 % of the cases that are missed by the JCTCK (see Fig. 5).

As there are only a few uncovered parts of the source code (see Table II), we will now explicitly discuss every one of them. The first condition that was not fully covered by our automatically generated test suite was a null pointer check. Some of the firewall functions perform a null pointer check before using the pointer. This condition is impossible for our test suite to cover, because our test suite is generated based on Section 6.2.8 of the JCRE specification and null pointers are not mentioned there. Therefore, no test case is generated targeting null pointers.

TABLE II: Conditions which were not fully covered

condition	JCTCK	our test suite
is object a null pointer	-	not to `true`
is object a global array	not to `true`	-
is object a shareable object	not to `false`	not to `false`
access of shareable object	not to `false`	not to `false`

A check if the accessed object is a global array is covered by our automatically generated test suite but not by the JCTCK: The JCTCK was not able to make the condition in the source code evaluate to `true`. The reason is that the condition is disjuncted with a check if the object is a temporary entry point object in the source code. As there is only one global array in the Java Card implementation, namely the APDU buffer, and this one is also a temporary entry point object, the short circuit evaluation in the C semantics renders it impossible to let the global-array-check evaluate to `true`. In contrast to that, our test adapter sets up the memory as required without the restrictions for a Java Card applet and was therefore able to generate an object which is no temporary entry point object but a global array.

Neither of the two test suites was able to make the checks regarding shareable objects evaluate to `false`. This is due to implementation specifics, which perform a check if the class or interface is shareable already in the implementation of the bytecode itself before calling the actual firewall function. Therefore, the firewall function is only entered if the condition evaluates to `true`. Note that in the implementation the firewall is not a monolithic function isolated from the rest of the code, but rather an optimized implementation taking advantages of available code.

So, in summary, the only parts of the firewall which are not covered by both test suites taken together are conditions that can only evaluate to fixed truth values due to checks that are made elsewhere in the code. Beside these conditions, full condition and basic block coverage is achieved on the code relevant for Section 6.2.8 of the JCRE specification.

D. Error Detection Results

All tests from the JCTCK relevant for the Java Card applet firewall passed with success. From our automatically generated test suite, however, the result of three test cases did not match the outcome of the oracle. Two of those were false positives: The firewall didn't deny access to objects with some attribute combinations because other parts of the implementation ensure that these attribute combinations can never occur. The third failing test case detected an actual inconsistency. It tested an access rule from Section 6.2.8.9, namely "*Otherwise, if the object is designated a Java Card RE Entry Point Object, access is allowed*". The result was a Security Exception, whereas the oracle expected that the access would be allowed because of this rule. An inspection of previous versions of the specifications confirmed that this access rule was added to the specification in version 2.2 [15], however, a review of the source code showed, that this modification of the specification was not implemented. Due to the limitations for object creation via Java Card applets it was also not possible to create a test case for the JCTCK which tests this behavior on the system level, so this inconsistency remained undiscovered until now.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

V. Conclusion and Future Work

In this paper, we have presented an automatic test case generation technique to achieve Modified Condition Decision Coverage on complex logical decisions. We implemented this approach in a test case generation tool using an SMT-solver to compute tests as satisfying assignments for the constraints that have to be satisfied to meet the coverage criterion. Our approach can handle complex couplings between different parts of the decision by delegating them to the SMT-solver. Our test case generation method can complement certification in the software development process by taking the existing models and closing the link from the requirements down to the implementation.

We evaluated our approach on an implementation of the Java Card operating system with focus on the applet firewall. Our tool produced only a small amount of test cases, but was able to improve the code coverage (condition + basic block coverage) of the existing test suite so that now all reachable locations and cases are covered. The additional tests produced by our tool also revealed that an update of the specification was not implemented. This confirms that different levels of testing are useful. System tests have to be complemented by unit and module tests because certain scenarios cannot (easily) be produced in the integrated system. The MCDC criterion proved to be effective in our setting because it tests the different parts of the decisions in isolation without producing too many test cases.

In the future, we plan to extend our test case generation approach and tool to deal with stateful models directly. This will relieve the user from writing a test adapter that brings the system into the desired state before the tests can be applied.

References

[1] P. Ammann, A. J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *International Symposium on Software Reliability Engineering (ISSRE'03)*. IEEE, 2003, pp. 99–107.

[2] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[3] G. Bernot, M. C. Gaudel, and B. Marre, "Software testing based on formal specifications: a theory and a tool," *Software Engineering Journal*, vol. 6, no. 6, 1991, pp. 387–405.

[4] G. Beuster and K. Greimel, "Formal security policy models for smart card evaluations," in *Annual ACM Symposium on Applied Computing (SAC'12)*. ACM, 2012, pp. 1640–1642.

[5] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 3 – Part 1: Introduction and general model*, July 2009.

[6] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 3 – Part 3: Security assurance components*, July 2009.

[7] B. Chetali and Q. H. Nguyen, "Industrial use of formal methods for a high-level security evaluation," in *International Symposium on Formal Methods (FM'08)*, ser. LNCS, vol. 2404. Springer, 2008, pp. 198–213.

[8] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," DTIC Document, Tech. Rep., 2001.

[9] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, 1994, pp. 193–200.

[10] A. Cimatti *et al.*, "NuSMV version 2: An opensource tool for symbolic model checking," in *Computer-Aided Verification (CAV'02)*, ser. LNCS, vol. 2404. Springer, 2002, pp. 359–364.

[11] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 1999.

[12] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[13] R. T. C. for Aeronautics (RTCA), "RTCA-DO-178B: Software considerations in airbone systems and equipment certification," Dec. 1992.

[14] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, Sep. 2009, pp. 215–261. [Online]. Available: http://dx.doi.org/10.1002/stvr.v19:3

[15] S. M. Inc., "Java Card TM 2.2 Runtime Environment (JCRE) Specification," 2006.

[16] G. Klein *et al.*, "seL4: formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, 2010, pp. 107–115.

[17] H. Mantel, W. Stephan, M. Ullmann, and R. Vogt, *Guideline for the Development and Evaluation of formal security policy models in the scope of ITSEC and Common Criteria Version 2.0*, December 2007.

[18] S. Motre and C. Teri, "Using B method to formalize the java card runtime security policy for a common criteria evaluation," in *National Information Systems Security Conference (NISSC'00)*, 2000.

[19] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software Testing, Verification and Reliability*, vol. 13, 2003, pp. 25–53.

[20] Oracle, "Java Card 3 Platform Runtime Environment Specification, Classic Edition Version 3.0.4," 2011.

[21] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31, no. 6, 1988, pp. 676–686.

[22] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a boolean specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, 1994, pp. 353–363.

Copyright (c) IARIA, 2013. **ISBN: 978-1-61208-307-0**

Enabling Interface Validation through Text Generation

Håkan Burden, Rogardt Heldal and Peter Ljunglöf
Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
E-mail: burden@cse.gu.se, heldal@chalmers.se, peter.ljunglof@cse.gu.se

***Abstract*—To obtain the information encoded in software it is necessary to master both the implementation languages and the tools. This is not only a problem for managers and user groups who have a claim in the outcome but not the necessary training in software development to decode the implementation - our case study shows that it is also a problem for the software developers. In contrast, text can be understood by everyone. The generation of textual summaries from software artifacts enables a more accessible format for validating software. Based on findings from interviewing practitioners in industry, we have developed a prototype for generating natural language summaries from component interfaces for validation purposes.**

Keywords—Natural language processing; Reverse engineering; Software components;

I. Introduction

One way of handling complexity in large-scale software development is to decompose the system into autonomous subsystems that can be independently developed and maintained. In order to successfully integrate the implemented subsystems into a complete and well-functioning system it is necessary to define the connecting points, the interfaces, of the subsystems before or during the implementation [1], [2].

However, the validation of the interfaces is not trivial. For code-centric development it requires an understanding of the used programming languages. In a model-based approach to software development the problem is described by Arlow et. al. [3] as consisting of three main challenges; the necessity to understand the modeling tools used during development, the need to understand and interpret the models that describe the subsystems and their interfaces as well as the underlying paradigm of the models. So, independent of how the subsystems and their interfaces are implemented their validation can only be done by those who understand the implementation. This excludes many of those that have a claim in the delivered system, such as managers and user groups but it also affects many of the system developers. In contrast, textual summaries have the benefit that they can be consumed by all with a claim in the developed software [4].

In an on-going study of model-driven software development for embedded systems in large corporations we discovered that the software engineers had problems with accessing the information encoded in the models in general as well as verifying the correctness of the interfaces in particular. To investigate the effort needed to generate textual summaries from the interfaces we developed a prototype solution, reusing existing software models.

Contribution: First we describe the problem of validating interfaces according to practitioners in industry and why NLG is a possible solution. We then show that the generation of textual summarisations of interfaces can be done with a limited additional effort. The natural language generation technique can be used for other interface specifications that belong to the same modelling language.

Overview: The next section presents the theoretical context of our study, while the practical details are given in Section III. The results are found in Section IV and we finish off with a discussion and possibilities for future explorations in Section V.

II. Theoretical Context

We begin by explaining a few theoretical aspects concerning software models and the specific methodology that we used for generating textual summaries. Related contributions in the area of generating textual summaries from software conclude this section.

A. Model-Driven Engineering

One of the aims of using software models is to raise the level of abstraction in order to capture what is generic about a solution. Such a generic, or *platform-independent* [5], [6], solution can be reused for describing the same software independent of the platform i.e. the operating system, hardware and programming languages that are chosen to implement the system. The level of detail in the models then depend on if they are to be used as *sketches* giving the general ideas of the system, *blueprints* for manual adaptation into source code or if they are *code generators* and developed with a tool that supports the automatic *model translation* into source code [7]. A condition for the latter is that there is a *metamodel* [5], [8] that specifies the syntactical properties of the modelling language, just as textual programming languages like Java and C have a syntactical specification that can be encoded in BNF [9], [10].

B. Executable and Translatable UML

Executable and Translatable UML, xtUML; [11], [12], evolved from merging the Schlaer-Mellor methodology [13] with the Unified Modeling Language, UML. Three kinds of

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

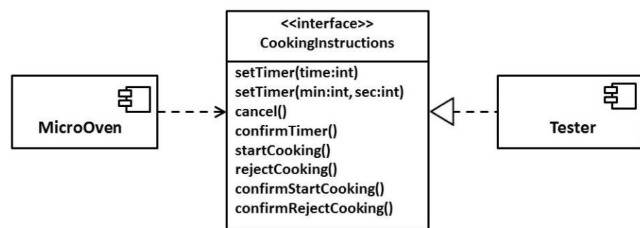


Fig. 1. The possible signals between the MicroOven and the Tester listed as an interface.

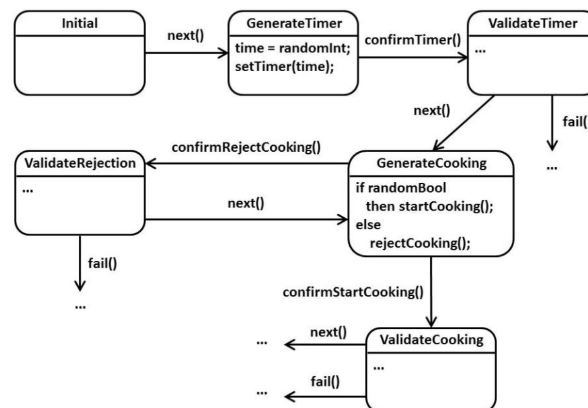


Fig. 2. The state machine in Tester describing the validation process.

graphical diagrams are used together with a textual *Action language*. The diagrams are *component diagrams*, *class diagrams* and *state machines*. The diagrams are organised hierarchically so that state machines are found inside classes, classes inside components and components can be recursively nested inside other components. The Action language is then used within the diagrams to specify their behaviour and properties. When the models are complete with respect to behaviour and structure they can be automatically transformed into source code through *model translation*.

Components: A component interacts with other components across an interface. An interface declares a contract in form of a set of public features and obligations but not how these are to be implemented. The information and behaviour of the component is only accessible through the specified interface so that the component can be treated as a black box. An example of two components and their interfaces is shown in Figure 1. It consists of two components, MicroOven and Tester where MicroOven provides an interface which Tester depends on.

Class diagrams: For the case of this study it is sufficient to view an xtUML class diagram as equivalent to a UML class diagram.

State machines: In the context of xtUML, a state machine is used to model the lifecycle of a class or an object [13]. The transitions between the states can either be defined as internal events or the signals defined by the interface can be mapped onto the transitions so that the external calls change the internal state of the component.

Figure 2 shows the lifecycle of a test object residing inside the Tester component. From the state *Initial* it is possible to reach the *GenerateTimer* state by the internal trigger *next()*. When the external trigger *confirmTimer()* is called the test object is updated and the new state is *ValidateTimer*. Some of the states include pseudo-code to indicate the action to be taken when entering that state.

The semantics of xtUML state machines differ from that of finite-state automata in that the former can interact with their environment as by creating and deleting instances of classes, dispatching events in other state machines and trigger the sending of signals across interfaces etc. Also, if a trigger event does not enable a transition it is not necessarily an error since transition triggers can be ignored if so desired.

Action language: An important property of xtUML is the Action language. It is a textual programming language that is integrated with the graphical models, sharing the same metamodel [13]. Since the Action language shares the same metamodel as the graphical models it can be used to define how values and class instances are manipulated as well as how the classes change their state. Thus we can find Action language within the operations of the classes but it is also used to define the behaviour and the flow of calls through the interface between the components.

Model translation: Code generation is a specific case of model translation with the aim of translating the model into code. However, model translations can just as well be used for reverse engineering the model into more abstract representations [14], [15]. Model translations are defined according to the metamodel, enabling the same transformations to be reused across domains [5], [16], just like a C compiler is defined on the BNF grammar, not on a specific C program [9]. The models become the code. An example of what the xtUML transformation rules look like and how they can be used is found in Figure 5, which will be further explained in section III-D.

C. Related Work

Previous research has reported on both formal and informal ways of validating the behaviour and structure of software. Examples of formal methods for validating the interfaces are presented by Hatcliff et. al. [17], Mencl [18] as well as Uchitel and Kramer [19] among many. However, they all have similar problems as those mentioned previously by Arlow et. al. [3]; formal methods require knowledge of the tools, knowledge of the used models and their paradigm as well as knowledge of the formal methods.

Lately there has been an increase in the attention towards more informal possibilities for validating software. Spreeuwenberg et. al. [20] argue that if you want to include all stakeholders in the development process you need to have a textual representation of the software models that has the right level of abstraction. In their case they generate a controlled natural language [21] to validate candidate policy decisions for the Dutch Immigration Office.

Another approach towards text generation from platform-independent representations is the translation between the Object Control Language, OCL [22], and English [23], [24].

This work was followed up by a study on natural language generation of platform-independent contracts on system operations [25], where the contracts were defined as OCL constraints that specified the pre- and post-conditions of system operations, i.e. what should be true before and after the operation was executed.

A crosscutting concern is a piece of functionality, such as an algorithm, that is implemented in one or more components. As a result of being scattered across the implementation they are difficult to analyse and when changed or updated it is difficult to estimate how the changes are going to affect the rest of the implementation. Rastkar et. al. [26] argue that having a natural language summary of each concern enables a more systematic approach towards handling the changes. They have therefor implemented a system for generating summaries in English from Java implementations.

Sridhara et. al. [27], [28] have also generated natural language from software implementations, in their case from Java code. The motivation is that understanding code is both a time consuming activity and that accurate descriptions can summarise the algorithmic behaviour of the code and as well as reduce the amount of code a developer needs to read for comprehension. The automatic generation of summaries from code mean that it is easy to keep descriptions and software synchronized. Another approach to textual summarisations of Java code is given by Haiduc et. al. [29]. They claim that developers spend more time reading and navigating code than actually writing it. Central to these publications is that they have to have some technique for filtering out the non-functional properties from the source code before translation into natural language.

There are two previous publications on generating textual descriptions from xtUML. The first describes how natural language specifications can be generated from class diagrams [30] while the second reports on the translation from Action language to English [31] e.g. these publications concentrate on generating textual summaries that describe the internal properties of the components instead of the interaction among components.

III. CASE STUDY

In our collaboration on model-driven engineering with Ericsson AB, Volvo Group Trucks Technology and Volvo Cars Corporation we encountered the problem of validating component interfaces. During our interviews the engineers reported that it was sometimes challenging to validate that the interface was correctly implemented and that the information needed for the validation could be difficult to obtain. As a response we developed a prototype to explore the possibilities to generate natural language summaries for validating component interfaces while keeping the added effort to a minimum.

A. Motivation

The interviews were conducted in January 2013, stretching into April 2013. The following interview extract illustrates the problem of understanding the implementation by reading its textual specification.

But we have a text document that's about 300 or 400 pages in total if you take all the documents. And that hasn't been updated for a couple of years. So this is wrong. This document is not correct.

Another issue is that sometimes the engineers are asked to specify the interface before they fully understand the internal behaviour of the component being developed. This means that defining the interface becomes guess work and subsequently there are signals that will never be used but still be given their share of the limited processing capacity.

Q: So do you overload the interface? Throw in a signal just in case?

A: Yes, that is what we do. At least I do it [. . .] and then you end up with the problem knowing which signal it is you should actually use.

One of the other interviewees had developed a work-around for handling that the interface specification was constantly outdated. The solution is to sieve through a second document after the information that concerns the interface being developed and translate that information into a new, temporary, specification.

We have in our requirements a list of signals used in the requirement. Now that list is seldom updated. It's hardly ever, so they're always out of date. So I don't actually read them anymore. I just go in through the specific sub-requirements and I read what is asked for my functionality. This is asked. What do I need? I need this and this. So, yeah, so I do it manually, I guess.

As a final example of the problems concerning the validation of the component interfaces, a software architect stated that the development tools were difficult to learn and that the development process would be much smoother if there was an accurate textual description of the implementation.

The tools are too unintuitive [. . .] the threshold for learning how to use them is high [. . .] but everybody knows how to consume text.

B. Aim

Generating text from the implementation should be a suitable solution since it allows the textual description to always be consistent with the implementation as well as understandable by all those with a claim in the project.

The aim of the text generation is a textual description of the intended usage of the interface with as little added effort as possible. For the generation to be feasible in an industrial setting it is beneficial if the generation rules can be maintained and updated without requiring new skills of the engineers. At the same time, the reuse of existing artifacts for generating the summaries will decrease their cost.

Two paragraphs are included in the generated text, one for the intended usage of the interface and one for the unused signals of the interface.

C. Setup

The generation is possible due to the reuse of an existing test model, that was adapted from Heldal et. al. [32]. They developed an executable test model for a microwave oven, as illustrated in Figure 1. The test model is designed to capture the intended dialogue between the MicroOven and the user, here

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

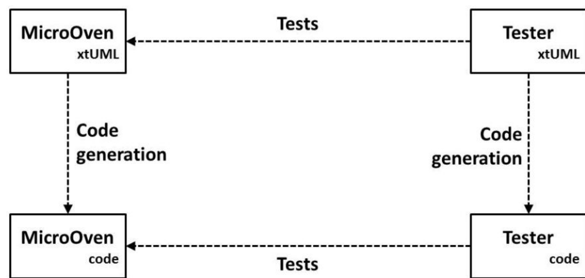


Fig. 3. The Tester-model can be reused at code level through code generation.

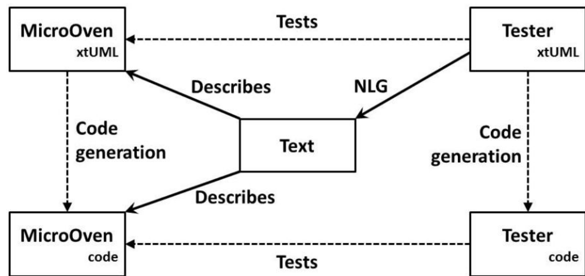


Fig. 4. The generated text describes both the xtUML model and the generated code.

```
01 .select many definedSignals related by interface -> C_EP[R4003] -> C_AS[R4004]
02 [...]
03 <table border="0">
04 <hr>Unused signals in MicroOven:</hr>
05 .assign unusedSignals = definedSignals - usedSignals
06 .if (not_empty unusedSignals)
07   .for each signal in unusedSignals
08     .invoke paramText = GetParamData
09     <tr><td><i>${signal.name}(paramText)</i></td></tr>
10   .end for
11 .else
12 <b>All defined signals are used.</b>
13 .end if
14 </table>
```

Fig. 5. An example of a model-to-text transformation using xtUML.

represented by the Tester component, as well as its possible error states and constraints. The sequence of the states and transitions therefor follows the process of the MicroOven, with additions for handling erroneous interactions. After the test case is initialised the test-pattern is to generate a signal to the MicroOven across the interface with random values for each parameter. The MicroOven's response is then validated before the Tester transitions into the *next* state in order to generate a new signal with random values. The test case needs to be able to store the results of prior interactions in order to compute the expected value in the validation states and compare that to the given response. If there is a mis-match the test case generates an internal *fail()* event and stores the resulting state so it can be diagnosticised.

After the MicroOven-model has been validated using the Tester-model they are both translated into platform-dependent source code. The MicroOven is then tested again, now as code by the Tester-code, to see that the intended behaviour of the oven remains the same after deployment. The relationship between the different representations of MicroOven and Tester are depicted in Figure 3.

D. Text Generation

Figure 5 shows a fragment of the generation rules. The rules are defined using the Rule Specification Language which are integrated with the xtUML tools [11]. On the first row the signals defined by the `interface` are selected by traversing the concepts of the metamodel according to their relationships. The concept `C_EP` refers to the executable properties of the interface and `C_AS` refers to those executable properties that are signals. The relationships between the concepts are referred to by the unique names `R4003` and `R4004`. Rows 3 and 4 show how the generated text is going to be physically represented [33] as html-pages, using a table since it enables the representation of parallel success paths. All rows that start with a punctuation mark are statements defined by the transformation language while those rows that do not start with punctuation mark define the generated text. The string value of a variable *v* is obtained by getting its literal text value, `${v}`, as in row nine where the signal's name is inserted into the table. Even if the success story only includes those signals that are implemented in the intended usage of the MicroOven the variable `usedSignals` on row five is defined by traversing the entire state machine in order to collect all signals that are used to implement the test case. On row eight the parameters are converted into a textual representation, `paramText`, by calling the function `GetParamData`, which is defined by the translation engineer.

In the context of our study it is not relevant to mention in what class the state machine resides that handles the interaction across the interface. That information is excluded in the content selection phase [34] since it is the possible interaction across the interface, as modelled by the state machine, that is interesting, not the internal structure of the components.

For the success path the structure of the text follows the order imposed by the transitions of the state machine, only considering the names of the transitions that constitute the intended usage.

IV. Results

The algorithm for navigating through the metamodel to generate the textual summaries is on the size of 100 statements. In comparison, a model compiler for generating Java programs consists of 500.000 statements but that covers the entire xtUML definition. Since the state machines and the Action language are so intertwined with the interfaces it is not possible to get a number for the statements needed for translating the interfaces as such into Java. The number of statements for the textual generation is dependent on the present content selection and will increase if more model concepts are to be present in the generation.

In Figure 6, an example of a generated text is shown. It depicts the summarisation of the interface in Figure 1 as implemented by the state machine in Figure 2. The top-half of the web page shows the intended usage of the oven and the bottom-half details which signals in the interface that are unused in the implementation. The intended usage is given in a table format where alternative usages are given on the same row, side-by-side.

The name of the interacting component is Tester, which is carried on to the generated text. This emphasizes that naming

Intended usage of MicroOven:
1: **Tester** sends *setTimer(time:int)*
MicroOven responds with *confirmTimer()*
2: **Tester** sends *startCooking()* 2: **Tester** sends *rejectCooking()*
MicroOven responds with *confirmStartCooking()* **MicroOven** responds with *confirmRejectCooking()*

Unused signals in MicroOven:
setTimer(min:int, sec:int)
cancel()

Fig. 6. An example of a textual summary.

conventions will affect the readability and understanding of generated texts when they are derived from software implementations. In this case the reading of the generated text would have benefited of naming the testing component to User, who it is meant to represent.

The paragraph for unused signals include *setTimer(min:int, sec:int)*, which represents how the interface was overloaded at the point of specification due to the fact that it was unclear how the MicroOven would be used. Since then the decision was taken to specify times in seconds only but the interface was not changed to reflect this decision. The generated text clearly identifies that there is a mismatch between the specification of the interface and its implementation.

Since the text is automatically generated from the source model it is possible to have a text generated that is consistent with the implementation whenever it is needed; e.g. when considering the implications of adding new functionality, to validate that new functionality conforms to the requirements during implementation or after implementation to understand how the software is intended to be used. This is shown in Figure 4 where the generated text can be used both to describe the original model or the implementation at code-level.

When the model is translated into code the information enclosed in the model is extended with details specific to operative system, chosen programming languages etc. in order to enable the deployment of the generated source code on a specific platform. This added information is then automatically excluded in the generated text since it is not present in the model. The benefit is that the generated text automatically becomes a summary of the interface that focuses on its intended usage while it abstracts away from how the behaviour is obtained. The text can then be reused independent on how the interface is realised on different platforms. As an example, the position of the signal *setTimer(time:int)* in the sequence of intended interactions between the oven and the user does not depend on if C or Java is used to realise the interface.

The algorithms for generating the success stories and to document unused signals are defined upon the xtUML metamodel. This means that they are reusable across different models that adhere to the metamodel, just as a compiler is defined upon the BNF grammar of programming language and therefor reusable across programs [9].

The structure and naming of concepts and relationships in the metamodel is the main source of complexity in this approach to NLG. Knowing what the concepts and relationships refer to is more challenging than how to map them into a textual representation.

V. Discussion and Future Work

The Object Management Group are the owners of the UML specification and the architects behind the MDA [5], [6] approach to using UML for software development. Their approach for defining the sequencing of the interface signals is to develop a new model, a protocol state machine[35]. Their solution results not only in an additional effort of developing a new model for explaining an old one, but also relies on the same techniques that made it difficult to verify the old model in the first place. As an additional contribution we show how an existing test model can be used for the same purpose as a protocol state machine as well as the source for an NL summary explaining the protocol of the interface.

In relation to previous work on text generation from xtUML our approach does not rely on the understanding of complex linguistic tools. The benefit of only using the same techniques for NLG as for code generation is that there is no additional training cost for companies. This makes it easier to adopt NLG in an industrial context since the number of software engineers with an understanding of both metamodelling and language technology are few. The mapping of metamodel concepts and relationships into linguistic properties will increase the complexity of macro- and microplanning. The drawback of our approach is the limited expressiveness of the transformation rules. For a more varied text structure, less repetitive sentences or for languages with a richer morphology it would be necessary to apply an NLG approach that incorporates linguistic competence. However, striking the balance between richer NLG and what companies are prepared to invest in hiring new competence is yet to be investigated. Our hope is that we will be able to start a new collaboration to enable practitioners in industry to evaluate both the generated texts and the generation procedure. Both lines of query would help to better understand where the balance between cost and readability lies.

Due to the time constraints of the MDE study there was not enough time to gain access to the original models to generate documentation within the industrial context where the issues were found. Instead, a prototype was implemented to show how the documentation could be generated from software models with a minimal effort. This opens for another possible route for the future, to further explore the possibilities of NLG for validation purposes in an industrial context. As seen in the related literature there is little involvement from industry to actually use NLG for validation purposes. We believe that this contribution can be a first step to address the problems that engineers actually are facing and as such also open for new ways of adapting NLG and summarisation techniques to the engineer's needs and context.

Acknowledgment

The authors would like to thank the Graduate School in Language Technology, the Center of Language Technology and the Software Center for funding and support as well as the contact persons at respective company for facilitating the study.

References

[1] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau, "Making Components Contract Aware," *IEEE Computer*, vol. 32, no. 7, pp. 38–45, 1999.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

[2] G. T. Heineman and W. T. Councill, Eds., *Component-based software engineering: putting the pieces together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[3] J. Arlow, W. Emmerich, and J. Quinn, "Literate Modelling - Capturing Business Knowledge with the UML," in *Selected papers from the First International Workshop on The Unified Modeling Language UML'98: Beyond the Notation*. London, UK: Springer-Verlag, 1999, pp. 189–199.

[4] D. Firesmith, "Modern Requirements Specification," *Journal of Object Technology*, vol. 2, no. 2, pp. 53–64, 2003.

[5] J. Miller and J. Mukerji, "MDA Guide Version 1.0.1," Object Management Group, Tech. Rep., 2003.

[6] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture™: Practice and Promise*. Addison-Wesley Professional, 2005.

[7] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2004.

[8] C. Atkinson and T. Kühne, "Model-driven development: a metamodeling foundation," *IEEE Software*, vol. 20, no. 5, pp. 36 – 41, September 2003.

[9] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston: Pearson Education, Inc., 2007.

[10] M. Wimmer and G. Kramler, "Bridging Grammarware and Modelware," in *Satellite Events at the MoDELS 2005 Conference*, ser. Lecture Notes in Computer Science, J.-M. Bruel, Ed. Springer Berlin / Heidelberg, 2006, vol. 3844, pp. 159–168.

[11] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie, *Model Driven Architecture with Executable UML™*. New York, NY, USA: Cambridge University Press, 2004.

[13] S. Shlaer and S. J. Mellor, *Object lifecycles: modeling the world in states*. Upper Saddle River, NJ, USA: Yourdon Press, 1992.

[14] T. Mens and P. V. Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, March 2006.

[15] P. Stevens, "A Landscape of Bidirectional Model Transformations," in *Generative and Transformational Techniques in Software Engineering II, International Summer School*, ser. Lecture Notes in Computer Science, R. Lämmel, J. Visser, and J. Saraiva, Eds., vol. 5235. Braga, Portugal: Springer, July 2007, pp. 408–424.

[16] S. J. Mellor, S. Kendall, A. Uhl, and D. Weise, *MDA Distilled*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[17] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath, "Cadena: An integrated development, analysis, and verification environment for component-based systems," in *Proceedings of the 25th International Conference on Software Engineering*, L. A. Clarke, L. Dillon, and W. F. Tichy, Eds. Portland, Oregon, USA: IEEE Computer Society, May 2003, pp. 160–173.

[18] V. Mencl, "Specifying Component Behavior with Port State Machines," *Electronic Notes in Theoretical Computer Science*, vol. 101, pp. 129–153, 2004.

[19] S. Uchitel and J. Kramer, "A workbench for synthesising behaviour models from scenarios," in *Proceedings of the 23rd International Conference on Software Engineering*, H. A. Müller, M. J. Harrold, and W. Schäfer, Eds. Toronto, Ontario, Canada: IEEE Computer Society, May 2001, pp. 188–197.

[20] S. Spreeuwenberg, J. Van Grondelle, R. Heller, and G. Grijzen, "Design of a CNL to Involve Domain Experts in Modelling," in *CNL 2010 Second Workshop on Controlled Natural Languages*, M. Rosner and N. Fuchs, Eds. Springer, 2010, pp. 175–193.

[21] A. Wyner, K. Angelov, G. Barzdins, D. Damljanovic, B. Davis, N. Fuchs, S. Hoefler, K. Jones, K. Kaljurand, T. Kuhn, M. Luts, J. Pool, M. Rosner, R. Schwitter, and J. Sowa, "On controlled natural languages: Properties and prospects," in *Proceedings of the Workshop on Controlled Natural Language (CNL 2009)*, ser. Lecture Notes in Computer Science, N. E. Fuchs, Ed., vol. 5972. Berlin / Heidelberg, Germany: Springer Verlag, 2010, pp. 281–289.

[22] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[23] R. Hähnle, K. Johannisson, and A. Ranta, "An Authoring Tool for Informal and Formal Requirements Specifications," in *FASE 2002, Fundamental Approaches to Software Engineering, 5th International Conference*, ser. Lecture Notes in Computer Science, R.-D. Kutsche and H. Weber, Eds., vol. 2306. Grenoble, France: Springer, April 2002, pp. 233–248.

[24] D. A. Burke and K. Johannisson, "Translating Formal Software Specifications to Natural Language," in *5th International Conference on Logical Aspects of Computational Linguistics*, ser. Lecture Notes in Computer Science, P. Blache, E. P. Stabler, J. Busquets, and R. Moot, Eds., vol. 3492. Bordeaux, France: Springer Verlag, April 2005, pp. 51–66.

[25] R. Heldal and K. Johannisson, "Customer Validation of Formal Contracts," in *OCL for (Meta-)Models in Multiple Application Domains*, Genova, Italy, 2006, pp. 13–25.

[26] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *27th International Conference on Software Maintenance*. Williamsburg, VA, USA: IEEE, September 2011, pp. 103–112.

[27] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52.

[28] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 101–110.

[29] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," in *WCRE*, G. Antoniol, M. Pinzger, and E. J. Chikofsky, Eds. IEEE Computer Society, 2010, pp. 35–44.

[30] H. Burden and R. Heldal, "Natural Language Generation from Class Diagrams," in *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, ser. MoDeVVa 2011. Wellington, New Zealand: ACM, October 2011.

[31] ——, "Translating Platform-Independent Code into Natural Language Texts," in *MODELSWARD 2013, 1st International Conference on Model-Driven Engineering and Software Development*, Barcelona, Spain, February 2013.

[32] R. Heldal, D. Arvidsson, and F. Persson, "Modeling Executable Test Actors: Exploratory Study Done in Executable and Translatable UML," in *19th Asia-Pacific Software Engineering Conference*, K. R. P. H. Leung and P. Muenchaisri, Eds. Hong Kong, China: IEEE, December 2012, pp. 784–789.

[33] J. Bateman and M. Zock, "Natural Language Generation," in *The Oxford Handbook of Computational Linguistics*, ser. Oxford Handbooks in Linguistics, R. Mitkov, Ed. Oxford University Press, 2003, ch. 15.

[34] E. Reiter and R. Dale, *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[35] "OMG Unified Modeling Language™(OMG UML), Superstructure," http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/, version 2.4.1. Accessed August 2013.

Efficient Elimination of False Positives Using Bounded Model Checking

Tukaram Muske, Advaita Datar, Mayur Khanzode, Kumar Madhukar
TRDDC, Tata Consultancy Services,
Pune, India
{t.muske, advaita.datar, mayur.khanzode, kumar.madhukar}@tcs.com

***Abstract*—Software verification using abstract interpretation is scalable but imprecise. Model checking is precise in verifying a property but not scalable. Often, these two techniques are combined to achieve better precision. A possible way is to analyze a software system first by using abstract interpretation and later eliminating the false positives using bounded model checking. This is a time consuming process as it typically involves verifying an assertion corresponding to each generated warning. We observe verifying all assertions often introduces redundancy, and some verifications may not even eliminate a false positive. In this paper, we present an approach consisting of three techniques to make such false positives elimination faster. Two of the techniques identify an assertion as being equivalent to an other assertion thus avoiding its verification. The third technique tries to identify and skip a class of assertion verifications that will not eliminate a false positive. Empirical results indicate that these techniques are quite useful in reducing the number of assertions being verified by 53%, and the false positives elimination time by 60%.**

***Keywords*—Abstract Interpretation; Model Checking; False Positives Elimination; Data Flow Analysis.**

I. INTRODUCTION

Software verification using abstract interpretation [1] has been effective in proving the absence of runtime errors such as Division by Zero (ZD), Array Index Out of Bound (AIOB) and buffer overflow. It has successfully been used to verify very large software systems, but generates too many false warnings, commonly referred to as false positives [2]. On the other hand, model checking is precise for property verification, but it often faces the state explosion problem as programs include unbounded-loops, recursions, complex data structures [3][4]. Under these circumstances, a property verification may not succeed and it is a concern.

There have been several attempts at combining these two techniques to improve precision [5][6][7], i.e., to generate fewer warnings which, in turn, would reduce the cost of their manual review. Abstract interpretation, being light weight, is used first and then generated warnings are processed by a model checker to eliminate false positives [8][9]. This processing includes generation of an assertion corresponding to each warning and its verification using a model checker. If an assertion is successfully verified, its corresponding warning is a false positive and is eliminated.

This process of False Positives Elimination (FPE) is very time consuming as each assertion needs to be verified by a model checker, and an average assertion verification time is significant [9][10]. Hence, there is a need to make it faster. To the best of our knowledge, nothing has been done to make such false positives elimination efficient. In practice, we observe verifying all assertions often introduce redundancies, and few verifications even may not eliminate a false positive. We use these observations to make FPE efficient by reducing the number of assertions being verified.

Throughout this paper:

1) we use A_n to denote the assertion at line *n* and $V(A_n)$ to denote its verification.
2) in our examples, for clarity of representation, we have eliminated some parts of the code (like assigning non-deterministic values to input variables, conditional pre-processor code to include single assertion at a time).
3) we describe FPE more particularly, using CBMC (C Bounded Model Checker) [11]. We chose CBMC for our experimentation because we have prior experience in its usage and it is integrated in the existing tool set [9].

Consider the motivating example in Figure 1, where each access of an array is reported as a warning by abstract interpretation (AIOB property), since their index values are unknown for abstract interpretation. This example also shows six assertions, one corresponding to each of the warnings. Verifications of all these six assertions is not required since some of them are redundant or non-verifiable. This is explained as below:

1) A_{15} and A_{17} are equivalent since their assert expressions are same and *n* is not modified in between. With this equivalence, $V(A_{17})$ is found redundant as it has same result as that of $V(A_{15})$. That is, if A_{15} is verified successfully, warnings related to A_{15} and A_{17} will be eliminated, else they will continue to be warnings after FPE.
2) If *Overflow-Underflow* (OFUF) property is proved on the code, $V(A_{21})$ has same result as that of $V(A_{23})$, because the involved *index* variable has an *unsigned* data type. That is, success in $V(A_{23})$ guarantees success in $V(A_{21})$, whereas if $V(A_{23})$ fails then $V(A_{21})$ is most likely to fail. Thus, their corresponding warnings will together be either false positives or continue to be warnings after FPE. This observation finds the $V(A_{21})$ is redundant.
3) $V(A_{35})$ and $V(A_{39})$ require analysis of the unbounded loop (run time dependent loop at line 34), and hence, these verifications either will terminate with *out of memory* or will produce an error trace depicting insufficient loop unwinding. Please refer to Section II-A for more details on insufficient loop unwinding by CBMC. These assertions

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

```
    int rColors[10], gColors[10];

11.   void f1(int r, int g){
12.      unsigned int i = 0, n, temp;
13.
14.      n = ...;
15.      assert( n>=0 && n<10 );
16.      rColors[n] = r;
17.      assert( n>=0 && n<10 );
18.      gColors[n] = g;
19.
20.     i = ...;
21.     assert(temp=i++, temp>=0 && temp<10 );
22.     rColors[i++] = ...;
23.     assert(temp=i++, temp>=0 && temp<10 );
24.     gColors[i++] = ...;
      }
```

```
    char *str, charArr[20];

31.   void f2(){
32.     int i = 0;
33.     ...
34.     while( *str != ' ' ){
35.         assert( i>=0 && i<20 );
36.         charArr[i] = *str;
37.         i++;
38.     }
39.     assert( i>=0 && i<20 );
40.     charArr[i] = '\0';
        ...
      }
```

Figure 1: Example and Annotation - 1

can not be verified, so are non-verifiable; we are unable to eliminate a false positive.

We propose three techniques for efficient FPE.

1) Property Independent Redundant Assertion Identification Technique (PI-RAIT): It partitions assertions and selects a leader assertion for each partition such that if leader assertion hold so does the other assertions from its partition. A partition so formed represents a set of either false positives or the warnings together, and it requires only the leader assertion to be verified. This technique will put A_{15} and A_{17} in the same partition with A_{15} as the leader.
2) Property Dependent Redundant Assertion Identification Technique (PD-RAIT): It includes partitioning of assertions similar in PI-RAIT, but depends on the characteristic of a run-time property. It only differs with PI-RAIT in the way of identifying equivalence of assertions, where it uses property characteristics and practical observations (code patterns) to identify the equivalence of assertions. This technique will put A_{21} and A_{23} in the same partition with A_{23} as the leader.
3) Non-Verifiable Assertion Identification Technique (NVAIT): It includes identifying assertion verifications which require analysis of unbounded loops and most likely they can not be verified successfully. This technique will identify A_{35} and A_{39} as *Non-Verifiable Assertions* (NVAs).

Out of six assertions in Figure 1, these proposed techniques identify only two assertions (A_{15} and A_{23}) to be verified and skip the other four. This makes FPE considerably faster. We applied this approach in different FPE settings for two automotive industry C applications and the results indicate that these techniques reduce the number of assertions being verified by 53% and the resultant FPE time by 60%.

We discuss in detail PI-RAIT in Section II, and PD-RAIT and NVAIT in Section III. The implementation and empirical results are described in Section IV. Section V presents related work, and finally, we conclude with future work in Section VI.

II. Property Independent RAIT

This section discusses in detail the proposed Property Independent RAIT (PI-RAIT) and presents an algorithm for it.

A. CBMC

In this paper, we describe FPE and the proposed techniques using CBMC, hence it is briefly described. CBMC is a Bounded Model Checker for ANSI-C and C++ programs. It takes an entry function and a property to be verified that is expressed as an assertion. The specified entry function represents a context at which the assertion is verified and can be an entry point of the application or any other function having the input assertion. If an assertion holds for all execution paths, CBMC reports verification success. If it does not hold, generates an error trace leading to the property violation. The verification is performed by unwinding the loops in the program, and it is necessary for all loops to have a finite upper bound [4]. For unbounded loops, it takes a user provided bound (unwinding count) as an upper bound. The provided bound should be enough to capture the program semantics, so that the property verification is sound and complete. Further, when the bound is not enough, it produces a trace (*loop unwinding counterexample*) to demonstrate the insufficiency of loop unwinding.

B. Reduction of Assertions in FPE: Need

Abstract interpretation usually reports a large number of analysis warnings [4]. In a FPE process, ideally, an assertion corresponding to each generated warning should be verified with application entry point as the entry function. However, it often does not scale or generates a loop unwinding counterexample. To overcome this problem, a different approach of incremental code context (*context expansion*) is adopted [8]. In this approach, an assertion verification is started with a minimal code context only, i.e., the enclosed function of the assertion. Later, the context is incremented to the callers of the enclosed function until one of the following holds:

1) its corresponding false positive is eliminated
2) context reaches the application entry
3) a certain time limit is achieved

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

FPE with code context expansion, as compared to FPEs at function and application levels, eliminates more false positives but involves verifying an assertion multiple times. Performing numerous verifications for each of the generated assertion increases FPE time even further. Hence, there is a need to minimize the number of assertions being verified.

C. Equivalence of Assertions

We have observed that, in practice, many of the generated warnings are similar, and hence, their corresponding assertions are also likely to be equivalent. The code snippet in Figure 2 depicts three Zero Division (ZD) warnings and their corresponding assertions. These warnings are similar, because the denominator in each ZD warning is the same variable taking values from the same source. They together represent a class of false positives or an error. Hence, the added assertions are also equivalent.

```
11.       denom = ...;
12.       if(...){
13.           assert(denom != 0); r1 = n1/denom;
14.       }
15.
16.      assert(denom != 0); r2 = n2/denom;
17.
18.       if(...){
19.           assert(denom != 0); r3 = n3/denom;
20.       }
```

Figure 2: Example and Annotation - 2

More precisely, two assertions are equivalent if -

1) their assert expressions are structurally similar *and*
2) the variables referred to by these assertions have the same source for their values.

The structural similarity of expressions requires that variables used, the operators and their order of appearance in the expression be the same. For example, given two ZD assertions with their expressions as

- $(a+b+c)! = 0$ and $(a+b+c)! = 0$ are potentially equivalent.
- $(v+1)! = 0$ and $(1+v)! = 0$ are not equivalent, since operands appear in different order.
- $(v1 + func())! = 0$ and $(v1 + func())! = 0$ are not equivalent since different calls to a function can return different values.

D. Partitioning of Assertions

We use the equivalence of assertions to reduce the number of FPE assertion verifications. The equivalent assertions are put in the same partition. An assertion in a partition is tagged as a *Leader Assertion* (LA) only if its successful verification by model checker guarantees the successful verification of other assertions in its partition. This ensures that the warnings corresponding to assertions in a partition are regarded as false positives when the leader of the partition is verified successfully. In the other case, if the leader verification fails then verification of other assertions in that partition is most likely to fail. Essentially, the verification result of assertions in a partition follow the verification result of the partition leader, hence, they are referred to as *Follower Assertions* (FAs). If an assertion can not be equivalent to an other assertion, it will be the only member (LA) of its partition without having the follower(s). Thus, in a partition there will be strictly only one LA and any number of FAs including zero.

Using this approach, burden of eliminating a false positive corresponding to a FA is transferred to its leader, and hence, there is a chance that it might miss on eliminating a false positive. This is because, there could be a scenario in which a FA is able to identify a false positive but its leader is not. We permit this assuming such scenarios would be very rare in practice.

We tag an assertion A in a partition as a LA, only if all paths reaching any other assertion (FA) also go through A. Under this criterion, A_{16} is selected as a LA for the partition of the equivalent assertions in Figure 2. Other assertion (A_{13} or A_{19}), can not be tagged as a LA since there exists a path reaching A_{16} but not going through it.

E. Assertions Partitioning Algorithm

We use *must reachability* and *must liveness* of assertions to compute the LAs and associate them with their corresponding FAs. The reaching and live assertions being computed denote *must* data flow information, i.e., they represent the assertions that are definitely reaching or definitely live from the program points at which they appear. These *must reaching (must live)* assertions are similar to *reaching definitions (live variables)* [12], with assertions replacing variables and *must* information replacing *may*. It should be noted that the assertions under consideration are unique in themselves where they are uniquely identified by the program points at which they appear.

1) Must Reaching Assertion (MRA): An assertion with expression e at a program point P is a MRA at its succeeding program point P' if every path coming to P' passes through P and, no path segment between P and P' contains a *l-value* occurrence of any of the *r-value(s)* of e. This ensures that each execution path through P' also includes P.

With a slight abuse of notation, we denote a program point of an assertion by the assertion itself. In Figure 2, A_{16} is a MRA at A_{19}. However, it is not a MRA at A_{13} since A_{13} appears before the A_{16}. Also, A_{13} is not a MRA at other two assertions since there exists a path that does not go through A_{13} but reaches to them (A_{16} and A_{19}).

2) MRAs Data Flow Formalization: We present the data flow formalization for MRAs computation at *procedure* level using forward information flow. It considers one run-time property at a time while computing MRAs, for example, MRAs for partitioning of ZD and AIOB assertions are computed separately. The equations below are shown for a node n in a *control flow graph* [13], where n denotes either an assignment statement or an expression controlling the flow.

$$In_n = \begin{cases} \emptyset & n \text{ is the entry node} \\ \bigcap\limits_{p \,\epsilon\, \text{predecessors}(n)} Out_p & \text{otherwise} \end{cases} \quad (1)$$

$$Out_n = Gen_n + (In_n - Kill_n(In_n)) \quad (2)$$

Copyright (c) IARIA, 2013.

$$Gen_n = \begin{cases} \{A\} & n \text{ has an assertion } A \\ \emptyset & \text{otherwise} \end{cases} \quad (3)$$

$$Kill_n(X) = \begin{cases} killInfo(X, n) & n \text{ modifies at least one variable} \\ \emptyset & \text{no variable is modified by } n \end{cases} \quad (4)$$

$$killInfo(X, n) =$$
$$\{A \in X \mid (usedVars(A) \cap modifiedVars(n)) \neq \emptyset\} \quad (5)$$

$$usedVars(A) = \textit{r-values} \text{ from assertion } A \quad (6)$$

$$modifiedVars(n) = \textit{l-values} \text{ from program statement } n \quad (7)$$

In the above formalization,

- In_n represents the MRAs flowing in to n, i.e., at the start of n, whereas Out_n represents the MRAs flowing out (at the exit) of n.
- In_n equation (1) uses intersection because the flow information being computed is a *must* information.
- information flowing in at a point is computed using the information flowing out of its predecessors. This is because we compute the flow information in the forward direction.
- MRAs information is generated only at program points having assertions while kill information is computed at each variable modification point.
- In_n equation (1) indicates that MRAs are computed at procedure level. This equation will need a change if the MRAs are to be computed at the application level. The change is required to incorporate the effect of calling contexts and function call points.

3) Must Live Assertion (MLA): An assertion with expression e at a program point P is a MLA at its preceding program point P' if every path coming out of P' also passes through P and, no path segment between P' and P contains a *l-value* occurrence of any of the *r-value(s)* of e. This ensures each execution path having P' on it, also includes P.

In Figure 2, A_{16} is a MLA at A_{13}. However, it is not a MLA at A_{19} since A_{19} does not precede A_{16}. Also, A_{19} is not a MLA at the other two assertion points (A_{16} and A_{19}) since there exists a path that does not go through A_{19} but reaches them.

4) Data Flow Formalization for MLAs: The formalization for MLAs computation will be similar to that of MRAs. The only change here is the direction of information flow which is *backward* in case of MLAs. In order to account for this change, we change the In_n and Out_n equations as under. Out_n and In_n denote the MLAs being computed at the exit and start of a program point n respectively.

$$Out_n = \begin{cases} \emptyset & n \text{ is the exit node} \\ \bigcap_{s \,\epsilon\, \text{successors}(n)} In_s & \text{otherwise} \end{cases} \quad (8)$$

$$In_n = Gen_n + (Out_n - Kill_n(Out_n)) \quad (9)$$

5) Computation of LAs and FAs: Once MRAs and MLAs are available at each program point of an application, identification of partitions with their associated LAs becomes easy. We denote the MRAs *at the exit* of a program point n (flowing out of n) as $MRAs(n)$, and the MLAs *at its start* (flowing in at n) as $MLAs(n)$. Assertions A and A', with their assert expressions as e and e' respectively, form elements in the same partition if e and e' are structurally similar and one of the following hold:

1) $A \in (MRAs(A') \cup MLAs(A'))$. In this case, A' will be a FA and A can be its leader.
2) $A' \in (MRAs(A) \cup MLAs(A))$. In this case, A will be a FA and A' can be its leader.

An assertion A can be selected as a leader of a partition if for every other assertion A' in the partition, $A \in MRAs(A') \cup MLAs(A')$. If more than one assertion in a partition qualify to be a leader, one of them is randomly selected as the leader.

F. Assertions Partitioning: Limitation

```
11.      denom = ...;
12.      if(...){
13.          assert(denom != 0); r1 = n1/denom;
14.      } else {
15.          assert(denom != 0); r3 = n3/denom;
16.      }
```

Figure 3: Limitation scenario of PI-RAIT

The usage of MRAs and MLAs to identify LAs sometimes does not partition the assertions which are equivalent but not definitely reachable from one another. Examples of this are A_{13} and A_{15} in Figure 3. In spite of being equivalent, they will not be partitioned together because there will not be a MRA or a MLA available at an assertion point from another assertion.

III. PD-RAIT AND NVAIT

This section discusses the details of PD-RAIT and NVAIT, and presents an algorithm for both.

A. PD-RAIT

We have observed that, often, there are large number of assertions whose verification result follows the verification result of some other assertion, and they do not fall under the same partition during PI-RAIT. For instance, in Figure 1, A_{21} and A_{23} will not be identified as equivalent by PI-RAIT, even though the result of $V(A_{21})$ follows that of $V(A_{23})$. We use this characteristic peculiar to AIOB warnings to reduce the number of assertion verifications by partitioning them in a way similar to PI-RAIT.

1) Partitioning of AIOB assertions: In Figure 1, the need is to identify A_{23} as a leader and A_{21} as its follower. This will require backward analysis. Therefore, we use MLAs (as in PI-RAIT) to partition the AIOB assertions. The only change here is in the way MLAs are computed. The rest of the algorithm

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

to identify the LA and its associated FAs remains the same as in PI-RAIT.

We describe the change required in MLAs computation using the same example in Figure 1. We denote the MLAs at the exit of a program point *n* as $Out(n)$, and the MLAs at the start of *n* as $In(n)$. In PI-RAIT, $A_{23} \in Out(A_{21})$ but $A_{23} \notin In(A_{21})$ because A_{23} gets killed at A_{21}. This kill of A_{23} omits the association of A_{21} (follower) with A_{23} (leader). In PD-RAIT, we avoid such a kill computation. With this change, $A_{23} \in In(A_{21})$ and using PI-RAIT algorithm these will get partitioned together with A_{23} as the leader.

We skip the data flow formalization of MLAs computation in this technique due to lack of space. It is to note that this algorithm does not consider the MRAs for their partitioning. This approach can not be applied to partition the assertions associated with the AIOB warnings including a pre/post unary decrement operator or a signed data-type variable in their indexes. It is possible to refine the PD-RAIT technique to handle these limitation scenarios, but we do not do it as such scenarios are very rare in practice. Also, we do not apply this technique for ZD as its warnings with such patterns are very rare. Further, it can not be applied to partition overflow-underflow assertions since the relationship in verification results of LA and FAs can not be guaranteed.

```
  int *ptr1, **ptr2;
  #define NULL 0

    void f(...) {
      ...
11.   ptr1 = *ptr2;
12.
13.   assert(ptr1!=NULL); arr[0] = *ptr1++;
14.   assert(ptr1!=NULL); arr[1] = *ptr1++;
15.   assert(ptr1!=NULL); arr[2] = *ptr1++;
    }
```

Figure 4: DNP Assertions Example

Figure 4 presents another example for the Dereference of a Null Pointer (DNP) to illustrate the application of the PD-RAIT algorithm is property specific. The PD-RAIT algorithm used to partition the AIOB assertions can not be used for partitioning of these DNP assertions because successful verification of $V(A_{15})$ does not guarantee the same for $V(A_{13})$. However, it can be observed that if A_{13} is verified successfully, the successful verification of $V(A_{14})$ and $V(A_{15})$ is ensured. Thus, these assertions can be partitioned together with A_{13} tagged as a LA. It is intuitive that, such leader identification includes forward analysis, and hence, MRAs should be used instead of MLAs.

B. Non-Verifiable Assertions Identification Technique

Verification of an assertion by CBMC includes analysis of a provided entry function and the functions that are called directly or indirectly from the entry function. It uses a provided bound (unwinding count) for the unbounded loops during their unrolling. In the absence of this input (unwinding count) CBMC keeps unrolling the loop and eventually out of memory. When the input bound is not insufficient, it results into an unwinding assertion counterexample. This verification does not contribute in eliminating a false positive and is needless.

In practice, an application includes unbounded loops whose bound is determined only at the run-time. We present a few examples of the unbounded loops as follows:

1) an infinite loop such as $while(1)$, $for(;;;)$.
2) a loop in which a bound variable in its terminating condition takes values from library system calls.
3) a loop whose terminating condition is run-time dependent like $*ptr! =$'\0' and the string(s) pointed by ptr gets its content during run-time through $fgets(ptr)$.

If an assertion is *control* or *data dependent* on any of the above unbounded loops, then it is a NVA. An assertion A is *control* or *data dependent* on a loop l if A is dependent on a statement belonging to l. In Figure 1, A_{35} and A_{39} are the NVAs. This is because, each of them is *control* and *data dependent* on the unbounded loop starting at line 34. We skip the verification of these NVAs to make FPE faster.

We use the following algorithm to compute the NVAs:

1) Identify a set of unbounded loops (denoted as L_{UB}) used in the application. *Loop termination analysis* [14][15] can be used for this purpose.
2) For each assertion A and an entry function f_e,
   a) Identify the loops in f_e on which A is *control* or *data dependent*. Program dependence graph [16] can be used for this purpose. We denote this set of loops as L.
   b) If $(L \cap L_{UB}) \neq \emptyset$ then A is a NVA in the context of f_e.

It must be noted that the identification of an assertion as NVA is always with respect to an entry function. Further, this approach might wrongly mark an assertion as a NVA even if it is not. This, in turn, may make the FPE imprecise.

IV. Implementation & Experiments

This section covers the implementation details and experiments performed for the proposed FPE.

A. FPE Implementation

We implemented the proposed techniques in TCS Embedded Code Analyzer (TECA) [17] to eliminate false positives from the analysis warnings generated by it. TECA is a static analysis tool to verify C source code. We used the framework shown in Figure 5 to implement the proposed techniques. A short description of each of the component in the framework is provided below.

1) Static Analyzer: A static analysis tool (TECA) that performs verifications for properties AIOB, ZD, DNP, OFUF, etc. on input C code and reports safe, unsafe and warnings program points.

2) Code Annotator: This component annotates the source code to generate assertion corresponding to each warning produced by a static analysis tool.

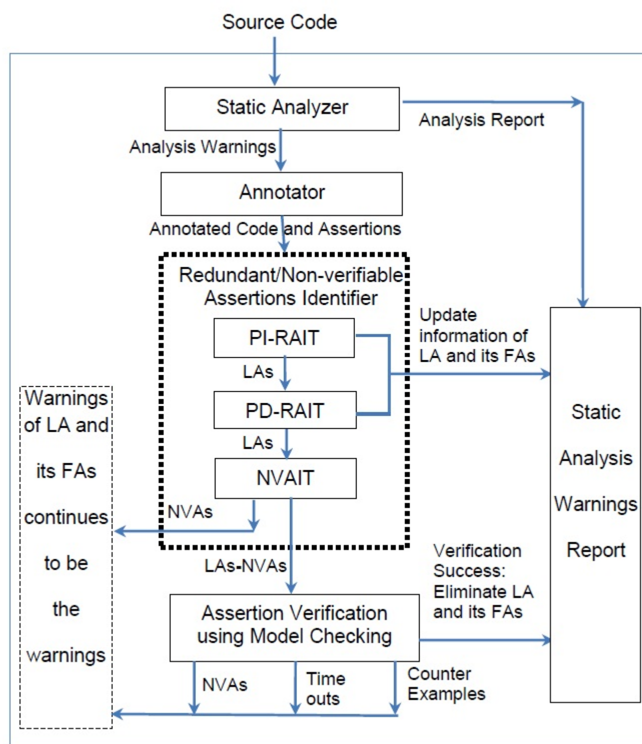


Figure 5: FPE Implementation Framework

3) Redundant/Non-Verifiable Assertions Identifier: We implemented the PI-RAIT, PD-RAIT and NVAIT as separate components, and use them in succession to get maximum benefit out of these techniques.

i. *PI-RAIT:* It implements the MRA and MLA formalizations in *context* and *flow* sensitive way at *function* level to partition the *input set* of assertions based on their equivalence. Further, it updates the analysis warnings report to reflect the association of LAs and their FAs.
ii. *PD-RAIT:* This component implements property specific PD-RAIT to partition the LAs computed by PI-RAIT. It does not analyze the FAs from PI-RAIT component since their analysis in this component would be redundant. Similar to PI-RAIT, it also updates the warnings report for its computed LAs and their FAs.
iii. *NVAIT:* It receives the PD-RAIT LAs and computes the NVAs from them. We used simple pattern based techniques to identify the unbounded loops in an application and did not use any complex loop termination analysis. In this component, due to lack of time, we used *must* reachability of unbounded loops instead of *control* or *data dependency*, to check if an assertion is a NVA.

4) Assertions Verifier: This component comprises mainly of a model checker (CBMC), and it optionally includes other tools implementing techniques such as code slicing [18], loops abstractions [9] to scale the model checker. The actual false positives elimination is performed by this component. It eliminates warnings (false positives) corresponding to a LA and its FAs when the LA is verified successfully. If the verification fails or times out, its corresponding warning is not eliminated. This component allows FPE in three different settings:

1) FPE at a *Function* level (F_{fpe}),
2) FPE at an *Application* level (A_{fpe}), *and*
3) FPE using a *Code context expansion* (C_{fpe}).

In C_{fpe}, assertions verifier component communicates with NVAIT component to check if the assertion being verified with an entry function is a NVA, and on finding the assertion as a NVA, its further verifications are skipped.

B. Experiments and Observations

We selected two embedded applications, one of 40 KLOC representing an automobile battery control module and another of 80 KLOC representing a smart card management system. Both the applications were verified for AIOB and ZD properties using abstract interpretation, and false positives were eliminated in three FPE settings - F_{fpe}, A_{fpe} and C_{fpe}. During our FPE experiments, we used-

1) 200 seconds time out for a CBMC verification.
2) sliced code with respect to generated assertion for its verification.
3) machine with Intel Core 2 Duo 2.33 GHz processor, 2 GB RAM configuration and having Windows XP SP3.

In Table I, we present details of the CBMC verification results for F_{fpe} and A_{fpe} settings without applying any of the proposed techniques (PI-RAIT, PD-RAIT and NVAIT). These results indicate that F_{fpe} outperforms A_{fpe} in terms of number of successful verifications. Also, there is a considerable number of unwinding counterexamples for these applications.

The results obtained for different combinations of the proposed techniques in each FPE setting are shown in Table II. For a FPE setting, it presents-

a. $|A_{in}|$, where A_{in} indicates a set of assertions those are verified by the CBMC.
b. $|E_{fp}|$, where E_{fp} is a set of false positives eliminated.
c. T_{fpe} representing the time taken ([Hours:Mins]) to verify assertions from A_{in}. It does not include the time spent in the code slicing.

In Table II, we also present the time taken in minutes (T_R) by a combination of the proposed techniques in a FPE setting. In our experiments, we applied PD-RAIT to AIOB only (and not for ZD). Following are the few observations from Table II.

1) In a setting, T_R is very less as compared to T_{fpe}, and this does not add any performance overhead in the FPE.
2) Among the FPE settings, the false positives eliminated are maximum in C_{fpe} and minimum in A_{fpe}.
3) Equivalent assertions found by PI-RAIT and PD-RAIT techniques, are more for AIOB compared to the ZD, and hence, the FPE time reduction is more for AIOB.
4) On an average, the PI-RAIT and PD-RAIT techniques have reduced 13.55% and 26.5% of FPE time respectively without compromising on the false positives eliminated. It indicates, although FPE with PI-RAIT and PD-RAIT is conservative in eliminating the false positives, in practice there is no miss on false positives eliminated.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

TABLE I: Distribution of CBMC Verification Results

Application	Property	Setting	$\|A_{in}\|$	CBMC Timeouts	Verification Successful	CBMC Trace	Unwinding Assertions	CBMC Failures
Battery Control Module	AIOB	F_{fpe}	430	13	107	238	71	1
		A_{fpe}	430	0	1	0	429	0
	ZD	F_{fpe}	47	6	5	16	20	0
		A_{fpe}	47	0	0	0	47	0
Smart Card Management System	AIOB	F_{fpe}	314	5	13	231	65	0
		A_{fpe}	314	42	0	0	250	0
	ZD	F_{fpe}	54	0	24	22	8	0
		A_{fpe}	54	26	12	0	7	9

TABLE II: FPE Experiment Results

Application	Property	Techniques	T_R	F_{fpe}			A_{fpe}			C_{fpe}		
				$\|A_{in}\|$	$\|E_{fp}\|$	T_{fpe}	$\|A_{in}\|$	$\|E_{fp}\|$	T_{fpe}	$\|A_{in}\|$	$\|E_{fp}\|$	T_{fpe}
Battery Control Module	AIOB	-	-	430	107	04:11	430	1	04:14	430	270	13:45
		PI-RAIT	≈ 1	301	107	03:01	301	1	03:03	301	270	10:37
		PI-RAIT + PD-RAIT	≈ 1	196	107	01:50	196	1	01:53	196	270	06:47
		PI-RAIT + PD-RAIT + NVAIT	≈ 3	157	105	01:22	7	1	00:11	157	265	05:29
	ZD	-	-	47	5	01:06	47	0	00:29	47	8	02:16
		PI-RAIT	≈ 1	37	5	01:02	37	0	00:29	37	8	01:39
		PI-RAIT + NVAIT	≈ 2	22	2	00:16	0	0	00:00	22	5	00:41
Smart Card Management System	AIOB	-	-	314	13	01:45	314	0	09:38	314	22	29:13
		PI-RAIT	≈ 1	229	13	01:17	229	0	07:34	229	22	22:53
		PI-RAIT + PD-RAIT	≈ 2	177	13	01:03	177	0	06:43	177	22	19:14
		PI-RAIT + PD-RAIT + NVAIT	≈ 2	165	12	00:59	116	0	03:53	165	20	19:03
	ZD	-	-	54	24	00:14	54	12	00:41	54	29	01:28
		PI-RAIT	≈ 1	53	24	00:14	53	12	00:41	53	29	01:25
		PI-RAIT + NVAIT	≈ 2	50	24	00:13	26	10	00:24	50	29	01:24

5) On an average, the NVAIT technique has reduced the FPE time by 38.91% with the identification of 32.54% of assertions as NVAs, but it has compromised on 1.3% of false positives. This indicates that NVAIT makes FPE efficient as well as imprecise.
6) The application of all the three techniques, on an average, reduces the $|A_{in}|$ and T_{fpe}, respectively by -
   - 53.37% and 61% in F_{fpe} setting.
   - 82.37% and 71.4% in A_{fpe} setting.
   - 53.37% and 43% in C_{fpe} setting.
7) NVAIT technique when applied in A_{fpe} setting to battery control module, found all the assertions as the NVAs. The reason was traced to the inclusion of typical $while(1)$ loop implemented in *main* function of an embedded application.

V. Related Work

There are a number of techniques that combine static analysis with model checking to improve its precision. These techniques differ in a way these two are combined. Brat et al. [5] do this in such a way that static analysis component iteratively exchanges information with the model checker. The partial order information computed by static analysis is used by model checker for its state space reduction, and the aliasing information from model checking is used to refine the results of static analysis. Fehnker and Huuck [6] analyze the counterexamples generated through model checking by using abstract interpretation to learn new facts and refine the abstraction. This continues until a warning is either proved to be a bug or spurious.

Rödiger [19] combines data flow analysis and model checking to improve the security vulnerability detection. The vulnerable code statements are found based on invalidated user inputs and they are model checked to eliminate false positives or produce a readable counterexamples. Junker et al. [7] present an abstraction refinement technique to automatically find and eliminate the false positives. It is achieved by iteratively computing the infeasible sub-paths using SMT solvers and refining the models. Wang et al. [20] and Tsitovich [21] present techniques among the others that combine static analysis and model checking.

The techniques described above, combining static analysis and model checking, focus only on improving analysis precision. Further, these are difficult to use when static analysis is performed by widely used commercial tools (like Polyspace and Coverity) since it needs changes in the tool's back-end analysis. Post et al. [8] and Darke et al. [9] try to overcome this limitation by using model checking to eliminate the false positives produced by the static analysis tools. Of these two techniques, [8] presents an approach to eliminate the false positives generated by Polyspace, where it uses incremental context expansion to do so.

A major drawback of these two techniques ([8][9]) is that they generate and verify an assertion corresponding to each static analysis warning and, hence, involve numerous verification calls to a model checker. While we also generate an assertion corresponding to each of the output warning, we avoid verifying each assertion. We partition the generated

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

assertions and verify only one representative assertion from a partition, so that all the false positives in the partition are eliminated at once. We employ two techniques - one dependent on the property being verified and the other independent of it. Further, we try to skip a class of non-verifiable assertions before we pass it to a model checker.

VI. Conclusion and Future Work

In our experiments, we have found an abundance of redundant assertions in FPE. Our techniques helped in minimizing the verification calls to a model checker and, in turn, made the FPE faster. The property-dependent and property-independent RAITs marked 45% of the assertions as the *followers*. This is because there are multiple equivalent assertions in certain code regions and they often fall under the same partition. Eliminating the false positive corresponding to the leader of the partition eliminates all the false positives corresponding to the followers as well. This allows us to skip the verification of followers. Although we eliminated false positives conservatively in our approach, it was never the case in our experiments that we failed at eliminating one. The results of PD-RAIT technique indicate that using code-pattern based approach to partition assertions can be quite useful in reducing the FPE time. This is because these patterns are widely used.

The identification of NVAs based on unbounded loops, using NVAIT, is quite effective in minimizing the FPE time (led to an average reduction of 38.91%). There are many unbounded loops in an application and the need to verify an assertion dependent on them gets eliminated. This approach, being conservative, may find an assertion as a NVA even if it is not. That is to say, it might wrongly mark a verifiable assertion as a NVA that could have possibly eliminated a false positive. This explains the trade-off between precision and performance of FPE using this technique.

Our experiments depict that the reduction in FPE time is dependent on property being verified and the context at which false positives are eliminated. Although the experiments are performed on embedded domain applications written in C, we expect similar benefits on other domain applications as well due to common coding practices. These techniques can be extended further to verify properties in applications coded in other languages too.

We plan to experiment further with NVAIT technique replacing CBMC by SATABS [22]. We are also exploring a technique to make FPE more efficient by identifying assertions whose verifications are more likely to generate counterexamples.

References

[1] P. Cousot and R. Cousot, "Basic concepts of abstract interpretation," in *Building the Information Society*, ser. IFIP International Federation for Information Processing, R. Jacquart, Ed. Springer US, 2004, vol. 156, pp. 359–366. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-8157-6_27

[2] D. Engler, "Concur 2005 - concurrency theory," M. Abadi and L. de Alfaro, Eds. London, UK, UK: Springer-Verlag, 2005, ch. Static analysis versus model checking for bug finding, pp. 1–1. [Online]. Available: http://dx.doi.org/10.1007/11539452_1

[3] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, no. 4, 2009.

[4] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2008.923410

[5] G. Brat and W. Visser, "Combining static analysis and model checking for software analysis," in *Proceedings of the 16th IEEE international conference on Automated software engineering*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 262–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872568

[6] A. Fehnker and R. Huuck, "Model checking driven static analysis for the real world: designing and tuning large scale bug detection," *Innovations in Systems and Software Engineering*, vol. 9, no. 1, pp. 45–56, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11334-012-0192-5

[7] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, "Smt-based false positive elimination in static program analysis," in *ICFEM*, 2012, pp. 316–331.

[8] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *ASE*, 2008, pp. 188–197.

[9] P. Darke, M. Khanzode, A. Nair, U. Shrotri, and R. Venkatesh, "Precise analysis of large industry code," in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 1, 2012, pp. 306–309.

[10] K. Vorobyov and P. Krishnan, "Comparing model checking and static program analysis: A case study in error detection approaches," in *International Workshop on Systems Software Verification (SSV'10)*, 2010.

[11] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[12] U. Khedker, A. Sanyal, and B. Sathe, *Data Flow Analysis: Theory and Practice*. Taylor & Francis, 2009. [Online]. Available: http://books.google.co.in/books?id=9PyrtgNBdg0C

[13] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/390013.808479

[14] A. Tsitovich, N. Sharygina, C. Wintersteiger, and D. Kroening, "Loop summarization and termination analysis," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, P. Abdulla and K. Leino, Eds. Springer Berlin Heidelberg, 2011, vol. 6605, pp. 81–95. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19835-9_9

[15] A. Bradley, Z. Manna, and H. Sipma, "Termination analysis of integer linear loops," in *CONCUR 2005 Concurrency Theory*, ser. Lecture Notes in Computer Science, M. Abadi and L. Alfaro, Eds. Springer Berlin Heidelberg, 2005, vol. 3653, pp. 488–502. [Online]. Available: http://dx.doi.org/10.1007/11539452_37

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[17] "TCS Embedded Code Analyzer (TECA)," [retrieved: October, 2013] http://www.tcs-trddc.com/trddc_website/scripts/project_detail.php?lab=SWRD&project_id=53,.

[18] A. D. Lucia, "Program slicing: Methods and applications," in *SCAM*, 2001, pp. 144–151.

[19] W. Rödiger, "Merging static analysis and model checking for improved security vulnerability detection," Masters, 2011. [Online]. Available: http://www.xn--wolfrdiger-icb.de/publication/roediger2011security.pdf

[20] L. Wang, Q. Zhang, and P. Zhao, "Automated detection of code vulnerabilities based on program analysis and model checking," in *SCAM*, 2008, pp. 165–173.

[21] in *Logic Programming*, ser. Lecture Notes in Computer Science, M. Garcia de la Banda and E. Pontelli, Eds., 2008, vol. 5366, pp. 822–823.

[22] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, ser. Lecture Notes in Computer Science, vol. 3440. Springer Verlag, 2005, pp. 570–574.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

State Space Reconstruction for On-Line Model Checking with UPPAAL

Jonas Rinast, Sibylle Schupp
Institute of Software Systems
Hamburg University of Technology, Hamburg, Germany
Email: {jonas.rinast,schupp}@tuhh.de

Dieter Gollmann
Security in Distributed Applications
Hamburg University of Technology, Hamburg, Germany
Email: diego@tuhh.de

***Abstract*—On-line system verification requires the efficient reconstruction of the state space a model checker generates. This paper proposes an approach to reconstruct the current state of models of real-time systems, implements it in the Uppsala and Aalborg model checker (UPPAAL) and thus renders on-line model checking in UPPAAL possible. On-line model-checking can be employed if parameters of models need to be adjusted to real-world values in case models are inaccurate. Applications include closed-loop patient monitoring and care taking as patient models commonly fail to accurately model all interactions in the human body and thus cannot provide good long-term estimates to ensure the patient's safety. We exploit use-definition chains in state space transformations to reduce the amount of reconstruction transformations. During testing the method reduced the amount of transformations by 42% on average over all experiments.**

Keywords*—*State Space Reconstruction; On-line Model Checking; UPPAAL

I. Introduction

Medical treatment facilities have grown to rely significantly on medical devices for monitoring and treatment. Most devices are still operated manually today and need to be configured, maintained, and supervised by a care taker. Recently, closed-loop monitoring and treatment of patients became a research topic as experience shows that human errors are prevalent. Patient-in-the-loop systems try to autonomously assess the patient's state using a monitoring device and if necessary treat the patient automatically, e.g., via a remote infusion pump. Such a system must clearly be shown to cause no harm to the patient. Safety must be ensured to prevent harm from the patient not only during normal operation but also in case emergency situations arise.

Model checking is a well developed technique to verify that a system model conforms to its specification and thus may be applied to show the safety of such system. However, to make meaningful conclusions about the system's behavior it is necessary to have detailed and accurate models of the individual components of the system. In the medical domain, the model checking approach is therefore severely hampered if the patient needs to be modeled accurately, e.g., to make estimates on a drug concentration in the patient. Generally, a patient model is likely to be inaccurate as the physiology of human beings is complex and varies between individuals, e.g. blood oxygen and heart rate depend on the patients condition. A generalized model will always miss individual characteristics. Patient-in-the-loop systems thus could be proved safe with such models but might still put patients at risk.

On-line model checking is a recent model-checking variant that relaxes the need for models to be accurate far into the future. On-line model checking provides safety assurances for short time frames only and renews these assurances continuously during operation. Appropriate models for the system thus only need to be correct for the short time frame they are used in. The renewal of safety assurances then is carried out on models adapted to the current system state to ensure the system's safety for the next time window. This on-line approach thus allows safety assessment at all times and provides means to react before safety violations occur.

A model adaptation step first needs to create an initialization sequence that recreates the previous model state before adjusting single values. The reconstruction is necessary to allow the simulation of the model to continue from the state it was interrupted in. This paper presents an automated state reconstruction approach for the Uppsala and Aalborg model checker (UPPAAL) that eliminates the need for custom reconstruction procedures for every application. The developed reconstruction method serves as a base for an on-line model checking interface with UPPAAL as the underlying verification engine.

Naively, the state space can be reconstructed by executing the same transition sequence that was used to create the state in the beginning. However, if the simulation has already run a significant time the executed transition sequence is likely to be long and only continues to grow over time. A more direct way to the desired state space is needed to keep the reconstruction process fast and on-line model checking feasible. For our reconstruction approach we adopted use-definition chains to eliminate transformations that have no effect on the final state space. Such transformations occur when their results are overwritten before they are read. Our reconstruction method has been applied to seven different test models. The method always correctly reconstructed the original state space while yielding a reduction of the executed transformations in the range from 23% to 84%.

Interestingly our on-line model checking interface could not only be used to automatically carry out on-line model checking. The interface also allows generic dynamic adaptation of model parameters and thus could be used with parameter learning algorithms or for calibrating the model.

The rest of the paper is organized as follows: Section II shortly introduces model checking, on-line model checking, and the model checker UPPAAL. Section III first provides necessary information on UPPAAL's state space and its transformations and then explains our reconstruction approach. Section IV presents our evaluation results. Section V gives an overview on related literature and, lastly, Section VI summarizes the paper and suggests further research.

II. On-Line Model Checking

This section shortly introduces model checking and its on-line variant, on-line model checking. The technique is shown by way of example using the model checker UPPAAL; for a formal specification of UPPAAL see [1].

Generally, the model checking approach explores the state space of the given system model in a symbolic fashion to check whether the state space satisfies certain properties. Such properties are mostly derived from a requirement specification for the system, e.g., one could check whether or not a certain system state is actually reachable. The modeling and property languages vary greatly depending on the model-checking tool. Tools for various programming languages coexist with dedicated tools that support their own modeling language. Dedicated tools often use finite state automata as a base formalism for their models. UPPAAL is such a well-established, dedicated model checking tool, which was jointly developed by Uppsala University, Sweden, and Aalborg University, Denmark [2], [3]. It is based on the formalism of timed automata: an extension of finite state automata with clock variables to allow modeling of time constraints. A finite state automaton defines a transition system by defining locations and edges that connect these locations. Edges are fired to execute a transition from one location to another. The system state in this case is the current location of the automaton and the possible valuations of the clock variables.

Figure 1 shows the example model that will be used to demonstrate the proposed state space reconstruction method. The model consists of three locations, *Init*, *Inv*, and *Count*, where *Init* is the initial location indicated by the double circle. The model uses two variables: x, a clock variable, and c, a bounded integer variable. Clock variables are special variables that synchronously advance indefinitely unless they are bounded by one or more invariants on the current locations. The location *Inv* has such an invariant, `x <= 2`, to bound the clock x, thus the value of x in *Inv* can be any value between its value when it entered the location and 2. The model has a single transition from the initial location to *Inv*. This transition is annotated with a guard, `x >= 3`, and an update, `x = 0, c = 0`. Guards are used to enable and disable edges depending on the current state. Here, the clock x needs to be greater or equal to 3 for the edge to be enabled. Only then can it be fired and a transition occurs. Indeed, as there is no invariant on x on the initial location the edge is enabled for values greater or equal to 3. Upon firing of the edge the update is executed: the clock x and the bounded integer c are both reset to 0. The edge from *Count* to *Inv* is nearly identical to the previous edge: when x is greater or equal to 3 the edge may be fired but x is reset to 1 instead of 0 and c is not modified. As a consequence, the value of x in *Inv* is between 0 and 2 when the location is first entered and between 1 and 2 on every subsequent visit. The transition between *Init* and *Count* has no guard and shows that an update may consist of a complex expression: the update `c = (c + 1) % 7` increases c by 1 modulo 7.

As explained in the introduction, model checking relies on accurate long term models. On-line model checking is a variant of classic model-checking that eliminates the need for such models and thus may be applied when such models are unavailable. It reduces the modeling error by periodically adjusting the current state to the observed real state, e.g., by setting a model value to the exact value measured by a sensor attached to a patient. For example, if we consider the model in Figure 1 one could assume that the counter variable c is modeling some patient's parameter. If that parameter in reality occasionally jumps the model is inaccurate and needs to be adjusted by setting c to the correct value. On-line model checking performs the adjustments and thus the jumps do not need to be modeled accurately. Note that errors may still be present in the system under on-line model checking but the method predicts them in advance to react to them. On-line model checking requires the model analysis to finish before the next update interval. Though the main work is done by the model checker the reconstruction still consumes some time, which our method reduces compared to the naive automatic reconstruction approach.

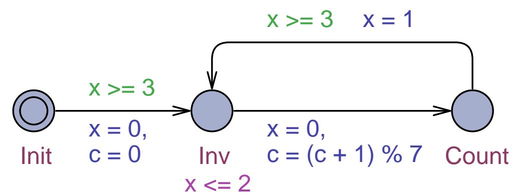


Figure 1. UPPAAL Model Example

III. State Space Reconstruction

In this section, we summarize the required information on UPPAAL's state space in Subsection III-A. Then, Subsection III-B presents our state space reconstruction process.

A. UPPAAL's State Space and its Transformations

UPPAAL's state space can be divided into three parts: the time state, the location state, and the data state. The location and the data state are straightforward: every data variable is assigned exactly one value for the data state and the location state consists of the current location vector, i.e., a vector that contains the current location for every automaton instance. The time state is more complicated as it needs to capture all possible valuations for every clock in the model as well as all relations between the clocks. *Difference bound matrices* (DBM) are a common and simple representation method for such time states [1], [4]. By introducing a static zero clock in addition to all the clocks in the model ($\mathcal{C}_0 = \mathcal{C} \cup \mathbf{0}$, $\mathcal{C}$ the set of all clocks) all necessary clock constraints can be written in the form $x - y \preceq n$ where x and y are clocks ($x, y \in \mathcal{C}_0$), $\preceq$ is a comparator ($\preceq \in \{<, \leq\}$), and n is an integer ($n \in \mathbb{Z}$). A value in a difference bound matrix then is a tuple of an integer and a comparator $(n, \preceq)$, $n \in \mathbb{Z}$, $\preceq \in \{<, \leq\}$ or the special symbol ∞, which indicates no bound. An order on the entries is given by $(n, \preceq) < \infty$, $(n_1, \preceq_1) < (n_2, \preceq_2)$ if $n_1 < n_2$, and $(n, <) < (n, \leq)$. Addition is defined as follows: $(n, \preceq) + \infty = \infty$, $(m, \leq) + (n, \leq) = (m + n, \leq)$, and $(m, <) + (n, \preceq) = (m + n, <)$. A difference bound matrix thus contains one bound, either including or excluding, for every pair of clocks $\mathbf{M} = (\{\mathbb{Z} \times \{<, \leq\}\} \cup \infty)^{|\mathcal{C}_0| \times |\mathcal{C}_0|}$.

As an example a clock constraint system with two clocks a and b and the constraints $a \in [2, 4)$, $b > 5$, and $b - a \geq 3$ is transformed to the canonical constraints $a - \mathbf{0} < 4$, $\mathbf{0} - a \leq -2$,

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

$b - \mathbf{0} < \infty$, $\mathbf{0} - b < -5$, $a - b \leq -3$, and $b - a < \infty$. The matching DBM is

$$\begin{array}{c|c|cc} & \mathbf{0} & a & b \\ \hline \mathbf{0} & 0 & (-2, \leq) & (-5, <) \\ a & (4, <) & 0 & (-3, \leq) \\ b & \infty & \infty & 0 \end{array}$$

During simulation of an UPPAAL model its transitions are repeatedly executed. Every transition generally has multiple effects on the time state and each such effect corresponds to a transformation of the difference bound matrix that represents the current time state. The following summary lists the DBM transformations necessary to traverse the state space [4]:

- *Clock Reset* A clock reset is performed when an edge is fired that has an update for a clock variable (`x = n`). A clock reset sets the upper and lower bound on the clock x to the given value and depending constraints, i.e., constraints on a clock difference involving x are adjusted. This corresponds to modifying the matrix row and column for the clock x.
- *Constraint Introduction* A constraint introduction is performed if either a firing edge has a guard on a clock or an invariant on a clock is present in a current location and the bound is more restrictive than the current constraint on the involved clock. In that case the relevant matrix entry is set to the new constraint and for all other entries in the matrix it is checked whether the new bound induces stricter bounds.
- *Bound Elimination* Bound elimination is performed when a new location is entered. All bounds on clock constraints of the form $c - \mathbf{0} < n$ are removed, i.e., the upper bounds on clocks are removed. Bound elimination is equivalent to setting the first matrix column except the top-most value to ∞.
- *Intersection* An intersection is performed if a state is constrained by multiple constraints. In that case all constraints are applied individually and their results are intersected to obtain the final result. Intersecting multiple DBMs is achieved by finding the minimum value for every matrix entry from all intersecting matrices.
- *Urgency Introduction* An urgency introduction is performed if an urgent or committed location is entered or an entered location has an outgoing, enabled transition that synchronizes on an urgent channel. Unlike the previous transformations, urgency is a modeling construct specific for UPPAAL to prevent time from passing. An urgency introduction is semantically equivalent to introducing a fresh clock on the incoming edge and adding a new invariant on that clock with a bound of 0 to the location. An urgency introduction thus can be derived from a clock reset and a constraint introduction.

Returning to the example model (Figure 1) the individual transitions can now be broken down into their respective transformations. The initial location *Init* induces a bound elimination on the initial state where all clocks are set to zero. The transition from *Init* to *Inv* yields a constraint introduction for the guard (`x >= 3`) and a subsequent clock reset (`x = 0, c = 0`). The reset of the bounded integer c is ignored here as c is part of the data state. The location *Inv* results in a bound elimination and a following constraint introduction to accommodate the invariant (`x <= 2`). The transition from *Inv* to *Count* simply induces a single clock reset transformation before the location *Count* eliminates the bound on the state space again. Lastly, the transition from *Count* to *Inv* introduces the same kind of transformations as the transition from *Init* to *Inv*: both perform a constraint introduction and a clock reset. The values computed for the clock variable x are as follows:

1) Location *Init*
   a) Initial: $x = 0$
   b) Bound Elimination: $x \in [0, \infty)$
2) Transition *Init* $\longrightarrow$ *Inv*
   a) Constraint Introduction: $x \in [3, \infty)$
   b) Clock Reset: $x = 0$
3) Location *Inv*
   a) Bound Elimination: $x \in [0, \infty)$
   b) Constraint Introduction: $x \in [0, 2]$
4) Transition *Inv* $\longrightarrow$ *Count*
   a) Clock Reset: $x = 0$
5) Location *Count*
   a) Bound Elimination: $x \in [0, \infty)$
6) Transition *Count* $\longrightarrow$ *Inv*
   a) Constraint Introduction: $x \in [3, \infty)$
   b) Clock Reset: $x = 1$
7) Location *Inv*
   a) Bound Elimination: $x \in [1, \infty)$
   b) Constraint Introduction: $x \in [1, 2]$

B. Reconstructing UPPAAL States

In many models a large number of previous transitions do not have an impact on the current state space. In the example model (Figure 1) this behavior can be observed: in the location *Count* the clock x is in the range $[0, \infty)$. This valuation was completely created by the clock reset of the ingoing edge and the bound elimination of the location itself. Previous state space transformations do not have any influence on the valuation of x. Therefore, instead of executing the transition sequence *Init* $\longrightarrow$ *Inv* $\longrightarrow$ *Count* totaling 7 transformations only 3 transformations are required. The introduction of a new initial state and the direct transition to *Count* with an update `x = 0` is sufficient to recreate the state space.

During reconstruction it is thus beneficial to exploit the fact that effects of certain state space transformations are overwritten by subsequent transformations. The key idea of our approach is the construction of use-definition chains to identify transformations that may be removed. A use-definition chain is a data structure that provides information about the origins of variable values: for every use of a variable the chain contains definitions that have influenced the variable and ultimately lead to the current value. Our idea is to adapt the definition-use chain technique from static data flow analysis on a program's source code to the state space reconstruction: every entry in the model's difference bound matrix is treated as variable and thus is observed for uses and modifications. DBM entries are

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

only modified by applying a state space transformation on the DBM. We thus analyzed the read and write access to matrix entries for every transformation to derive the use-definition chains where the transformations are the basic operations.

In the following, our reconstruction approach is presented using the clock reset transformation as a leading example. First, we discuss the derivation of use-definition chains from the transformation. Then we lay out the use of reference counters as a memory structure to identify removable transformations. Lastly, we give a short overview on model synthesis based on the shortened transformation sequence. Altogether the adaptation of the use-definition chain approach to the reconstruction process results in the following steps:

1) *Initialization* Canonize model by introducing general starting points for later synthesis, extract necessary information from the model.
2) *Simulation* Select transitions in the model according to intended behavior, execute and store them. Simultaneously break them down into matching state space transformations and apply them internally to construct the use-definition chains of the transformations. Remove unnecessary transformations on-the-fly using reference counters for the transitions derived from the use-definition chains.
3) *Synthesis* Group the sequence of reduced transformations to form transitions and add the transitions to a newly created model obtained from the original one. Match the last transition to the current location state and update the data state on that transition.

Starting points for the reconstruction algorithm are the algorithms for the original transformations. Figure 2 lists as an example an algorithm for the clock reset transformation on the difference bound matrix D according to Johan Bengtsson [4]. Examination of the algorithm yields that all values in the row and column that are associated with the reset clock are written and all values in the top-most row and left-most column are read: lines 3 and 4 of the algorithm write D_{ij} and D_{ji} and read D_{i0} and D_{0i}. Note that index j is always greater than 0 as it is a real clock and not the **0**-clock. Therefore, the reset transformation creates a use for every value in the top-most row and left-most column and a definition for every value in the row and column for the clock in question. Taking into consideration that the value D_{jj} will always evaluate to zero and no definition needs to be generated we construct a modified algorithm that captures the definitions and uses generated by the transformation. Figure 3 shows the modified algorithm. It has an additional parameter T, which is a matrix that contains the transformations that are responsible for the current DBM values. Additionally to the functionality of the original algorithm the new algorithm updates this matrix and creates the necessary definition and use information: in lines 6 and 7 we store that the reset transformation uses the transformations T_{0i} and T_{i0} and lines 8 and 9 update the matrix to show the reset transformation is now responsible for the values D_{ji} and D_{ij}.

We designed the transformation matrix data structure for the use-definition chains to allow on-the-fly removal of unnecessary transformations: as soon as a transformation is overwritten in the matrix a following transformation cannot have a dependency on that transformation. Thus, the transformation may

```
1: procedure RESET(D, x_j = m)
2:     for i ← 0, n do
3:         D_ji ← (m, ≤) + D_0i
4:         D_ij ← D_i0 + (−m, ≤)
5:     end for
6: end procedure
```

Figure 2. Reset Transformation Algorithm [4]

```
1: procedure RESET(D, T, x_j = m)
2:     for i ← 0, n do
3:         if i ≠ j then
4:             D_ij ← D_i0 + (−m, ≤)
5:             D_ji ← (m, ≤) + D_0i
6:             Use(T_0i)
7:             Use(T_i0)
8:             T_ji ← this
9:             T_ij ← this
10:        end if
11:    end for
12: end procedure
```

Figure 3. Modified Reset Transformation Algorithm

be directly removed if no intermediate transformation depends on it. If intermediate transformations exist the transformation can be deleted as soon as those are removed. To accurately track needed transformations the data structure uses reference counters: every transition is assigned a counter to indicate how often it is referenced and every transformation updates this counter. The benefit of the on-the-fly removal is reduced memory usage for the data structure and shorter processing time during transformation execution.

We analyzed all relevant DBM transformations for their reads and writes and adapted the algorithms to update the transformation matrix and the reference counters accordingly. Special attention had to be given to the intersection algorithm: if two transformations are applied to a DBM and only one of them writes a certain value but the previous value is the stronger bound the transformation that did not modify the matrix is responsible for the resulting entry. This behavior needs to be introduced during the intersection algorithm as the original transformation only creates relations for entries it can potentially modify. Also the reference counters have to be propagated accordingly. We encapsulate read-write relations of transformations in special linker objects such that other transformations may influence them later on and the effects of the transformation on the data structure may be deferred to appropriate times to manage the reference counters.

The synthesis of the actual UPPAAL model from the calculated transformation sequence has to take into consideration that UPPAAL allows a single automaton to be instantiated multiple times with possibly different parameters. During initialization of the reconstruction we therefore analyze the model definitions for automaton instantiation and save the relevant parameters. Also, as the location space needs to be correctly reconstructed an automaton that is instantiated multiple times has multiple initializations transitions for every instantiation. We use a single bounded integer variable in conjunction with appropriate guards to correctly order these transitions. Another important aspect of the synthesis step is that the model initialization needs to be self-contained, i.e., the ini-

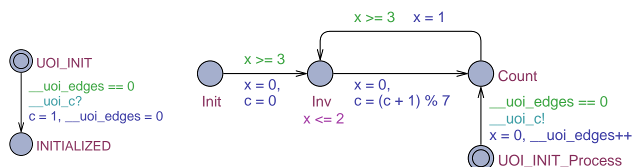


Figure 4. Reconstructed example model

TABLE I. EVALUATION RESULTS

Model	Transitions			Transformations		
	Before	After	Reduction	Before	After	Reduction
2doors	100	65.89	34.1%	364.7	254.46	30.2%
bridge	100	68.21	31.8%	188.39	144.09	23.5%
train-gate	100	66.18	33.8%	320.09	214.17	33.1%
fischer	100	91.27	8.7%	345.33	249.46	27.7%
csmacd2	100	100	0%	709.71	434.19	38.8%
csmacd32	75.58	75.58	0%	1818.6	327.49	79.7%
tdma	100	68.16	31.8%	719.88	240.11	66.6%
2doors	1000	627.9	37.2%	3722.3	2612.9	29.8%
bridge	1000	641.3	35.9%	1882.8	1436.4	23.7%
train-gate	1000	606.1	39.4%	3200.1	2194.1	31.4%
fischer	1000	853	14.7%	3455.3	2486.8	28%
csmacd2	1000	1000	0%	7238.1	4375.5	39.5%
csmacd32	619.6	619.6	0%	22491.1	2540.3	84%
tdma	1000	663.1	33.7%	6446.3	2651.5	58.9%

tialization of multiple automata needs to finish synchronously to prevent parts of the model from advancing prematurely. As the initialization transitions per automaton may differ in length we employ a broadcast channel to synchronize the last transition to the original model. We use these final transitions to initialize the data variables as well. In case global variables are present an additional init automaton is introduced for their initialization. Figure 4 shows the reconstruction model (right) for the example model (Figure 1) after 2 transitions. The additional initialization automaton (left) sets the global, bounded integer c to 1. The clock x is set to 0 and the location is correctly initialized to *Count* after an initial first transition. The reconstructed model only needs to execute a single transition in contrast to the original model, which uses two, to reach the correct state. For transformations the reconstructed model uses 3 time state transformations while the original model needs 7.

IV. EVALUATION

We evaluated our use-definition reconstruction method by applying it to seven different UPPAAL models and comparing it to the naive reconstruction approach. The models *2doors*, *bridge*, *train-gate*, and *fischer* are part of the UPPAAL example model suite. The *csmacd* models and *tdma* were taken from case studies. We ran two test sets for every model. The first test executed 100 times 100 random transitions of the model before reconstructing the state. The second test set executed 1000 random transitions 10 times. For the *csmacd32* model it was not always possible to execute the maximum number of transitions during simulation as the model exhibits deadlock states. Table I shows our evaluation results. In the top half the results of the first test set and in the bottom half the results of the second test set are shown. All values are averages over the respective test runs but their variances are small. In our experiments the reduction of transformations is between 23% and 84% while the reduction of transitions is between 0% and 39.4%. This difference mainly stems from the fact that to delete a single transition all induced transformations need to be removed. However, our model synthesis algorithm still is unoptimized and sometimes produces unnecessary transitions. In cases where the transition reduction is higher than the transformation reduction the removal of transformations made it possible to merge multiple transitions. Interestingly, the *cs-macd* models contain use-definition chains spanning the whole simulation, which prevent removal of transitions though many transformations are irrelevant to the state. Future work will need to address this issue, e.g., by also evaluating concrete state values. Regarding total execution time, our adjustments have a small impact as the model checking procedure consumes most of the time. Also, compared to the model checking part our approach scales well with the complexity of the used models.

V. RELATED WORK

The on-line model checking approach our reconstruction method is complementing and thus closest to has recently been proposed by Li et al. [5], [6]. They employ a hybrid automata model to ensure correct usage of a laser scalpel during laser tracheotomy to prevent burns to the patient. Yet, the necessary model initialization and reconstruction step is a custom solution and is not presented in detail. In the context of on-line model checking with UPPAAL, the UPPAAL variant UPPAAL Tron has been developed [7]. UPPAAL Tron is an on-line testing tool that can generate and execute test cases on-the-fly based on a timed automata system model. While the tool focus lies on input/output testing using a static system model the fact that the underlying model is an UPPAAL model means that our reconstruction approach might be beneficial for tests when the system model is inaccurate or still needs to be developed. Other related work falls in two categories: different ways to use or implement on-line model checking, and different ways to optimize state space exploration and representation in model checkers.

Qi et al. propose an on-line model checking approach to evaluate safety and liveness properties in C/C++ web service systems [8]. Their focus lies on consistency checks for distributed states to debug a system from known source code. Reconstruction is not an issue because the source code is static during execution. Easwaran et al. use a control-theoretic approach to the general runtime verification problem [9]. They introduce a steering component featuring a model to predict execution traces. Their approach uses a fixed prediction model while our reconstruction is for adapting inaccurate models. Sauter et al. address the prediction of system properties using previously gathered time series of measurements, e.g., taken by sensors [10]. They propose a split into an on-line and an off-line computation and to precompute expensive parts of the prediction step to reduce on-line work load. While their scenario of adapting using sensor measurements is applicable to our medical scenario with inaccurate patient models they focus on the verification load problem while we address the model inaccuracy. Harel et al. propose usage of model checking during the behavior and requirement specification step during development. Instead of interactively guiding the system to derive requirements a model checker executes the model and generally finds more adequate requirements. While

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

their approach employs on-line model checking their goal thus lies on early requirement development. In contrast our approach is useful in adaptation of deployed systems to ensure safety. Arney et al. present a recent patient-in-the-loop case study for automatic monitoring and treatment where UPPAAL and Simulink models were developed to verify safety questions beforehand [11]. They monitor heart rate and blood oxygen levels of the patient and automatically control drug infusion via a remote pump. On-line model-checking could benefit this scenario as currently a generalized patient model is employed and drug absorption rates may vary per patient.

Alur and Dill introduced timed automata and the underlying theory in 1994 [12] and Yi et al. developed the first implementation of the model-checker UPPAAL shortly after [13]. Many improvements have been made to the model-checking approach over the years. Larsen et al. proposed symbolic and compositional approaches to reduce the state-space explosion problem [14]. Partial order reduction on the state space was employed by Bengtsson [15]. Larsen et al. reduced memory usage on-the-fly using an algorithm that exploits the control structure of models [2], [16]. Further memory reductions were achieved by Bengtsson et al. with efficient state inclusion checks and compressed state-space representations [17]. Behrmann et al. provide an overview on current functionality and the usage of UPPAAL [3]. They also provide a more detailed presentation of UPPAAL's internal representations [18]. For a summary on timed automata, the semantics, used algorithms, data structures, and tools see [1]. Bengtsson's PhD thesis provides more in-detail information on difference bounded matrices [4].

VI. Conclusion and Future Work

In this paper we addressed the problem of state reconstruction of UPPAAL models in the context of on-line model checking. Our reconstruction method uses use-definition chains to track influence of individual transformations on the state space during model simulation. With the chains constructed and the additional use of reference counters we are able to identify and remove transformations in the transformation sequence that do not have an impact on the final state space. A prototype implementation was developed and compared to the naive reconstruction approach, which does not remove any transformations. Seven UPPAAL models from different sources were analyzed and our approach reduced the amount of transformations necessary for reconstruction by 23% to 84% and reduced model transitions by up to 39.4%.

In general, the proposed reconstruction method still yields infeasible reconstruction sequences for real-time on-line model checking as the reconstruction sequence length still grows over time. A reconstruction sequence of constant length is desirable to ensure real-time properties. Future research thus could focus on further optimizing the proposed reconstruction method. For example, the proposed method currently only relates transformations according to read and write accesses. Concrete variable values are not taken into account. Transformations that produce the same values could be removed, but are currently not. Experience during development has shown that such transformations occur often especially in periodic use-definition chains that arise due to cycles in the model. Removal of them could improve the reconstruction sequence significantly by breaking such cycles.

References

[1] J. Bengtsson and W. Yi, "Timed Automata: Semantics, Algorithms and Tools," in Lectures on Concurrency and Petri Nets, J. Desel, W. Reisig, and G. Rozenberg, Eds. Springer Berlin Heidelberg, 2004, ch. 3, pp. 87–124.

[2] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Efficient verification of real-time systems: compact data structure and state-space reduction," in Real-Time Systems Symposium, San Francisco, CA, USA, 1997, pp. 14–24.

[3] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal 4.0," Department of Computer Science, Aalborg University, Aalborg, Denmark, Tech. Rep., 2006.

[4] J. Bengtsson, "Clocks, DBMs and States in Timed Systems," Ph.D. dissertation, Uppsala University, 2002.

[5] T. Li et al., "From offline long-run to online short-run: Exploring a new approach of hybrid systems model checking for mdpnp," in 2011 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP 2011), 2011.

[6] T. Li et al., "From Offline toward Real-Time: A Hybrid Systems Model Checking and CPS Co-design Approach for Medical Device Plug-and-Play (MDPnP)," in Proceedings of the 3rd ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '12. Beijing, China: IEEE, April 2012, pp. 13–22.

[7] A. Hessel et al., "Testing real-time systems using UPPAAL," in Formal Methods and Testing, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Springer Berlin Heidelberg, 2008, pp. 77–117.

[8] Z. Qi, A. Liang, H. Guan, M. Wu, and Z. Zhang, "A Hybrid Model Checking and Runtime Monitoring Method for C++ Web Services," in 2009 Fifth International Joint Conference on INC, IMS and IDC. Seoul, South Korea: IEEE, 2009, pp. 745–750.

[9] A. Easwaran, S. Kannan, and O. Sokolsky, "Steering of Discrete Event Systems: Control Theory Approach," Electronic Notes in Theoretical Computer Science, vol. 144, no. 4, 2006, pp. 21–39.

[10] G. Sauter, H. Dierks, M. Fränzle, and M. R. Hansen, "Light-weight hybrid model checking facilitating online prediction of temporal properties," in 21st Nordic Workshop on Programming Theory, NWPT 09, vol. 2, Lyngby, Denmark, 2009.

[11] D. Arney et al., "Toward patient safety in closed-loop medical device systems," in Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '10. Stockholm, Sweden: ACM New York, NY, USA, 2010, pp. 139–148.

[12] R. Alur and D. L. Dill, "A Theory of Timed Automata," Theoretical Computer Science, vol. 126, no. 2, 1994, pp. 183–235.

[13] W. Yi, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving," in 7th International Conference on Formal Description Techniques, D. Hogrefe and S. Leue, Eds., 1994, pp. 223–238.

[14] K. G. Larsen, P. Pettersson, and W. Yi, "Compositional and symbolic model-checking of real-time systems," in Real-Time Systems Symposium, Pisa, Italy, 1995, pp. 76–87.

[15] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, "Partial order reductions for timed systems," in CONCUR'98 Concurrency Theory, D. Sangiorgi and R. de Simone, Eds. Springer Berlin Heidelberg, 1998, pp. 485–500.

[16] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Compact Data Structures and State-Space Reduction for Model-Checking Real-Time Systems," Real-Time Systems, vol. 25, no. 2-3, 2003, pp. 255–275.

[17] J. Bengtsson, "Reducing memory usage in symbolic state-space exploration for timed systems," Department of Information Technology, Uppsala University, Uppsala, Sweden, Tech. Rep. May, 2001.

[18] G. Behrmann et al., "UPPAAL Implementation Secrets," in Formal Techniques in Real-Time and Fault-Tolerant Systems, W. Damm and E.-R. Olderog, Eds. Oldenburg, Germany: Springer-Verlag Berlin, 2002, pp. 3–22.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

Formal Composition Based on Roles within a Model Driven Engineering Approach

Cédrick Lelionnais
Jérôme Delatour
and Matthias Brun
ESEO-TRAME
Angers, FRANCE
{cedrick.lelionnais,jerome.delatour,matthias.brun}@eseo.fr

Olivier H. Roux
and Charlotte Seidner
IRCCyN - Université de Nantes
École Centrale de Nantes
Nantes, FRANCE
{olivier-h.roux,charlotte.seidner}@irccyn.ec-nantes.fr

***Abstract*—Faced with the increasing complexity of Real-Time Embedded Systems, Model Driven Engineering offers the possibility of developping frameworks in which transformations are used to generate either executable code or formal models. However, these transformations themselves are generally not formalized. Correctness of transformations could therefore be called into question. This paper proposes a formalization of a transformation step, namely: the composition of formal fragments describing the behavior of a real-time system. These fragments are described using an extension of the classical Time Petri Nets, where the notion of roles was added to perform the composition of the fragments. This formalization increases confidence in transformations.**

Keywords*—*Model Driven Engineering, Real-time operating systems, Behavioral modeling, Transformation, Verification, Time Petri Nets, Application deployment

I. INTRODUCTION

Real-Time Embedded Systems (RTESs) increasingly surround us in various domains (aircrafts, automotive sector, cell phones, robotics, etc.). RTES engineers are confronted with the challenge of developing more complex, higher quality systems, with shorter development cycles at lower costs. Model Driven Engineering (MDE) [7] helps engineers to develop frameworks for partially automating the development of RTESs. Thanks to transformations, those frameworks produce either executable code or formal models from high-level descriptions of RTESs.

However, many frameworks do not consider the description of Real-Time Operating Systems (RTOSs) [4]. RTOSs have indeed an impact on the behavior of RTESs. In addition, in spite of the fact that the behavior of RTOSs starts to be considered, transformations have often been implemented within frameworks without formalization. Correctness of the transformation could therefore be called into question. Confidence in those frameworks could also be reduced.

The general approach presented in this paper aims to create a formal model of the whole system deployed on a RTOS. This approach was thought regardless of the intended RTOS. To do this, a transformation process is currently in progress to compose several behavioral fragments, each one describing a part of this system. Those fragments come from a model of the targeted RTOS, which is considered through the process execution. However, composition rules must be chained in a right sequence in order to avoid any ambiguity. As a basis of the construction, the use of roles formally identify connection points, which will be used as a glue of the system parts.

This paper is divided into the following sections. Section 2 refers briefly to the frameworks chosen for this contribution. The latter is presented in Section 3, highlighting both the deployment process and the use of roles. Section 4 deals with Time Petri Nets (TPNs) as translation formalism. A new syntax is then defined formalizing the composition of TPNs based on roles. Relying on this definition, Section 5 formalizes the construction of an application deployment in TPN. Consequently, the benefits and limits of this approach are discussed in Section 6. Finally, we conclude in Section 7.

II. RELATED WORKS

A first presentation of related works in conjunction with the consideration of RTOSs has already been presented in a previous contribution [4]. For this reason, frameworks in line with the consideration of RTOSs will only be presented here.

We have opted for frameworks in which the intervention of each stakeholder has been made more flexible. Indeed, the domain skills (RTOSs structure, transformations, deployments choice, etc.) are correctly separated with these frameworks. This has been made possible thanks to an explicit approach, which consists in considering each RTOS description without modifying the transformation rules. This strategy offers the possibility to capitalize most descriptions in a generic way. Furthermore, other works [3] [4] are based on the behavioral consideration of RTOSs. These contributions search for refining models of applications deployed on RTOS with the aim of verifying properties.

We can note the MARTE UML profile [8] in which the Software Resource Modeling (SRM) approach [12] has been integrated. With SRM, RTOSs can be modeled using stereotyped concepts from the real-time software domain. For another example, Software Execution Platform Inside Tools (SExPIsTools) is involved in the tooling of development processes. Real-Time Embedded Platform Modeling Language (RTEPML) [2] was developped in this sense, with the aim of defining concepts dedicated to the real-time domain for modeling RTOSs.

However, as introduced previously, the processes implemented in those frameworks have not yet been formalized. We have therefore decided to carry on developping SExPIsTools by experimenting the formalization.

III. CONTEXTUALIZATION IN A MDE APPROACH

This section is divided into two parts. The first part presents the SExPIsTools framework and the language RTEPML [2]. This presentation details the notion of role, which is used for the composition of behavioral descriptions. The second part specifies the language adopted for formalizing.

Copyright (c) IARIA, 2013. **ISBN: 978-1-61208-307-0**

A. SExPIsTools Framework

SExPIsTools (Figure 1) allows to generate code from high-level descriptions [2] to several RTOSs. The RTOS description (i.e., the Platform Description Model) is a parameter of the generic transformation made possible thanks to the notion of role. A role explicitly establishes a relationship between abstract concepts of RTOSs (i.e., the notion of task), their properties (priority of task) and their services (creation or destruction of task). The transformation rules rely on both concepts and roles. For each Platform Description Model, translation of roles is given in the Application Programming Interface (API) of the targeted RTOS. The descriptions are realized by the modeling language RTEPML.

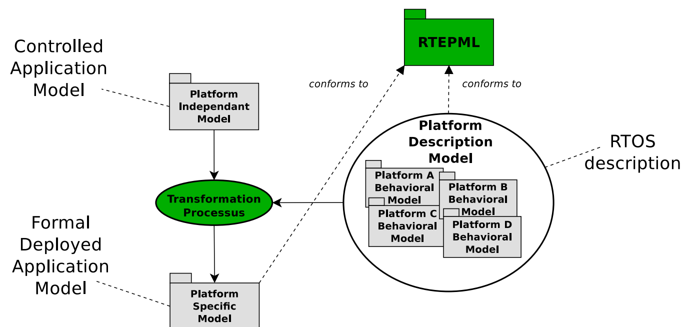


Fig. 1: SExPIsTools Process within MDE Context

RTEPML has been extended [4] to describe RTOSs behavior in a formal way. The purpose of formalizing was to allow model-checking. For each concept and service of the RTOS, a formal description (called fragment) is given. The transformation process leads to the composition of fragments. To facilitate this composition, some roles were added.

B. Choice of formal language

We have chosen TPNs [6] [1] to translate behavioral fragments because we need a formalism which expresses models with synchronism and parallelism for multitasking. Lastly, RTOSs imply time constraints. The chosen formalism needs to have clocks to represent time evolution.

IV. TPN Composition based on Roles

In order to compose fragments in TPN, we have projected roles on those fragments. To perform the composition, we have decided to assign roles to places. The interest of such a method is to merge places [11] [10], which are the connection points of the system that must be modeled in TPN.

In this section, TPNs with roles are firstly defined. The definition of the instanciation of TPN with roles is then given. Finally, the composition of TPNs is highlighted through a synchronization formalism based on roles.

A. TPNs

TPNs are a timed extension of classical Petri nets. Informally, to each transition of the net is associated an implicit clock and an explicit time interval. The clock measures the time since the transition has been enabled and the time interval is interpreted as a firing condition: the transition may fire if the value of its clock belongs to the time interval.

Definition 1 (TPN): A TPN is a tuple $\mathcal{T} = \langle P, T, \mathsf{Pre}, \mathsf{Post}, m_0, I_s \rangle$ where:

- P is a finite non-empty set of *places*;
- T is a finite non-empty set of *transitions*;
- $\mathsf{Pre} : P \times T \to \mathbb{N}$ is the *backward incidence* function;
- $\mathsf{Post} : P \times T \to \mathbb{N}$ is the *forward incidence* function;
- $m_0 : P \to \mathbb{N}$ is the *initial marking* of the net;
- $I_s : T \to \mathbb{N} \times (\mathbb{N} \cup \{+\infty\})$ assigns a *static time interval* to each transition.

A marking of $\mathcal{T}$ is an application from P to $\mathbb{N}$. Let m be a marking of $\mathcal{T}$. Then, for any place $p \in P$, we say that p contains $m(p)$ *tokens*. A transition $t \in T$ is said to be enabled by the marking m if $\forall p \in P, m(p) \geq \mathsf{Pre}(p,t)$. This is denoted by $t \in \mathit{enabled}(m)$. For any interval I_s, we denote by $I_s^{\downarrow}$ the smallest left-closed interval with lower bound 0 that contains I_s. For each transition t there is an associated clock x_t. We consider valuations on the set of clocks $\{x_t | t \in T\}$ and we will slightly abuse the notations by writing $v(t)$ instead of $v(x_t)$.

Let m be a marking of the net and t a transition in $\mathit{enabled}(m)$. Let m' be the marking obtained from m by firing t. Let m'' be the *intermediate marking* defined by $\forall p, m''(p) = m(p) - \mathsf{Pre}(p,t)$. A transition t' is *newly enabled* by the firing of t from m, and we note $t \in \uparrow\mathit{enabled}(m,t)$ if $t' \in \mathit{enabled}(m') \setminus \mathit{enabled}(m'') \cup \{t\}$

The operational semantics of the TPN $\mathcal{T} = \langle P, T, \mathsf{Pre}, \mathsf{Post}, m_0, I_s \rangle$ is defined by the time transition system $\mathcal{S}_{\mathcal{T}} = (Q, q_0, \to)$ such that:

- $Q = \mathbb{N}^P \times \mathbb{R}^T_{\geq 0}$
- $q_0 = (m_0, \mathbf{0})$
- $\to \in Q \times (T \cup \mathbb{R}_{\geq 0}) \times Q$ is the transition relation including a discrete transition and a continuous transition.
  - The discrete transition is defined $\forall t \in T$ by $(m, v) \xrightarrow{t \in T} (m', v')$ iff:
    - $t \in \mathit{enabled}(m)$;
    - $\forall p \in P, m'(p) = m(p) - \mathsf{Pre}(p,t) + \mathsf{Post}(p,t)$;
    - $v(t) \in I_s(t)$;
    - $\forall k \in [1, |T|], \quad v'_k(t_k) = \begin{cases} 0 \textbf{ if } t_k \in \uparrow\mathit{enabled}(m,t) \\ v_k(t_k) \textbf{ otherwise} \end{cases}$
  - The continuous transition is defined by $(m, v) \xrightarrow{d \in \mathbb{R}_{\geq 0}} (m, v + d)$ iff $\forall t' \in \mathit{enabled}(m), \forall 0 < d' \leq d, (v + d')(t') \in I_s^{\downarrow}(t')$.

Definition 2 (TPN with roles): A TPN with roles is a tuple $\mathcal{N} = \langle \mathcal{T}, R, \lambda \rangle$ where:

- $\mathcal{T}$ is a TPN,
- R is a finite set of roles,
- $\lambda : P \to R \cup \{\perp\}$ is the function assigning a role to a place and $\perp$ denoting that no role is assigned to a

place. Hereafter, some notations and properties of this function are enumerated :

1) $P_\lambda = \{p \in P \mid \lambda(p) \neq \perp\}$ is the set of places with role.
2) $\lambda_{\setminus P_\lambda} : P_\lambda \rightarrow R$ is an injective function;
3) $\lambda^{-1} : R \cup \{\perp\} \rightarrow P \cup \{\emptyset\}$ such that
$$\begin{cases} \forall r \in R,\ \lambda^{-1}(r) = \begin{cases} p \text{ if } \lambda(p) = r \\ \emptyset \text{ otherwise} \end{cases} \\ \lambda^{-1}(\perp) = \emptyset \end{cases}$$

The operational semantics of the TPN with roles $\mathcal{N} = \langle \mathcal{T}, R, \lambda \rangle$ is the same as that of TPN. Indeed, the use of roles within the definition of TPN does not impact its semantics.

B. Instanciation of TPN with roles

As seen previously, some fragments of TPN are instanciated before being composed. In order to distinguish the fragments to compose, all roles in a same fragment must be renamed according to the name of the instance.

Let $\mathcal{N}$ be the TPN to instanciate and x the label given to the instance. The renaming function $\rightharpoondown$ is a function from R to R_i where assigned roles are involved in parameters.

Definition 3 (Instanciation of TPN with roles): The instanciation of $\mathcal{N}$ with m renamings is denoted by

$$\mathcal{N}_i = Ins(\mathcal{N}, x) = \mathcal{N}\Big|_{\substack{r^1 \rightharpoondown r^1_x \\ \dots \\ r^m \rightharpoondown r^m_x}}$$

with $m = |R|$, $\forall j \in [1, m]$, $r^j \in R$, $r^j_x \in R_i$ and $\forall k \in [1, m]$, $k \neq j \rightarrow r^k \neq r^j$

C. TPNs Synchronization based on roles

In order to synchronize some TPNs, we must precise the definition of the composition of TPNs, which will be based on roles assigned to places. Let $\mathcal{N}_1, \dots, \mathcal{N}_n$ be n TPNs with $\mathcal{N}_i = \langle P_i, T_i, \mathsf{Pre}_i, \mathsf{Post}_i, m_{0_i}, I_{s_i}, R_i, \lambda_i \rangle$ such that $\forall k \neq k' \in [1, n] \implies T_k \cap T_{k'} = \emptyset$ and $P_k \cap P_{k'} = \emptyset$. The composition $\mathcal{N} = \langle P, T, \mathsf{Pre}, \mathsf{Post}, m_0, I_s, R, \lambda \rangle$ of the previous TPNs with roles will be denoted by $\mathcal{N} = \mathcal{N}_1 || \mathcal{N}_2 || \dots || \mathcal{N}_n$. Linked to this composition, we define a function leading to the merging of places whose assigned roles will be taken into account in parameters.

The merging function $\hookrightarrow$ is a partial function from $(R_1 \cup \{\bullet\}) \times (R_2 \cup \{\bullet\}) \times \dots \times (R_n \cup \{\bullet\}) \rightarrow P \times R$ where $\bullet$ is a special symbol used when a TPN is not involved in a particular merge of the global system. We then extend the definition of the assigning inverse function with $\lambda^{-1}(\bullet) = \emptyset$

The composition of n TPN with m merging is denoted by

$$\Big(\mathcal{N}_1 || \dots || \mathcal{N}_n\Big)\Big|_{\substack{(r_1^1, \dots, r_n^1) \hookrightarrow (p^1, r^1) \\ \dots \\ (r_1^m, \dots, r_n^m) \hookrightarrow (p^m, r^m)}}$$

with $\forall i \in [1, n]$, $\forall j \in [1, m]$, $r_i^j \in R_i$, $r^j \in R$ and $p^j \in P$, and $\forall k \in [1, m]$, $k \neq j \Rightarrow r_i^k \neq r_i^j$

We will subsequently use the following notations:

- Let $P_i^{merged} \subseteq P_i$ be the set of places of the net $\mathcal{N}_i$ merged by the composition. Formally $P_i^{merged} = \bigcup_{\forall j \in [1,m]} \{\lambda_i^{-1}(r_i^j)\}$
- Let $P^{\hookrightarrow} \subseteq P$ be the set of places of the net $\mathcal{N}$ obtained by the merging. Formally $P^{\hookrightarrow} = \bigcup_{\forall j \in [1,m]} \{p^j\}$

Definition 4 (Composition of TPNs with roles): The composition of the n TPN $\mathcal{N}_i$ with the merging $\hookrightarrow$ denoted by:

$$\mathcal{N} = \Big(\mathcal{N}_1 || \dots || \mathcal{N}_n\Big)\Big|_{\substack{(r_1^1, \dots, r_n^1) \hookrightarrow (p^1, r^1) \\ \dots \\ (r_1^m, \dots, r_n^m) \hookrightarrow (p^m, r^m)}}$$

is defined by:

- $R = \Big(\bigcup_{\forall i \in [1,n]} \big(R_i \setminus \bigcup_{\forall j \in [1,m]} \{r_i^j\}\big)\Big) \cup \Big(\bigcup_{\forall j \in [1,m]} \{r^j\}\Big)$;
- $P = \Big(\bigcup_{\forall i \in [1,n]} P_i \setminus P_i^{merged}\Big) \cup P^{\hookrightarrow}$;
- $T = \bigcup_{\forall i \in [1,n]} T_i$;
- $\lambda : P \rightarrow R$ is defined by:
  - $\forall p \in P \setminus P^{\hookrightarrow}$ meaning that $\exists i$ such that $p \in P_i$ then $\lambda(p) = \lambda_i(p)$
  - $\forall p^j \in P^{\hookrightarrow}$, meaning that p is the result of a merging, $\lambda(p^j) = r^j$
- $\mathsf{Pre} : P \times T \rightarrow \mathbb{N}$ is defined $\forall p \in P$ and $\forall t \in T_i \subseteq T$ by
$$\mathsf{Pre}(p,t) = \begin{cases} \mathsf{Pre}_i(p,t) \text{ if } p \in P \setminus P^{\hookrightarrow} \text{ and } p \in P_i \\ \mathsf{Pre}_i(p',t), \text{ if } \begin{cases} p \in P^{\hookrightarrow} \text{ and } p' \in P_i \\ (\dots, r_i^k, \dots) \hookrightarrow (p, \lambda(p)) \\ \lambda_i(p') = r_i^k \end{cases} \\ 0 \text{ otherwise.} \end{cases}$$
- $\mathsf{Post} : P \times T \rightarrow \mathbb{N}$ is defined $\forall p \in P$ and $\forall t \in T_i \subseteq T$ by
$$\mathsf{Post}(p,t) = \begin{cases} \mathsf{Post}_i(p,t) \text{ if } p \in P \setminus P^{\hookrightarrow} \text{ and } p \in P_i \\ \mathsf{Post}_i(p',t), \text{ if } \begin{cases} p \in P^{\hookrightarrow} \text{ and } p' \in P_i \\ (\dots, r_i^k, \dots) \hookrightarrow (p, \lambda(p)) \\ \lambda_i(p') = r_i^k \end{cases} \\ 0 \text{ otherwise.} \end{cases}$$
- $m_0 : P \rightarrow \mathbb{N}$ is defined $\forall p \in P$ by:
$$m_0(p) = \begin{cases} m_{0i}(p) \text{ if } p \in P \setminus P^{\hookrightarrow} \text{ and } p \in P_i \\ \sum_{i=1}^{n} m_{0i}\big(\lambda^{-1}(r_i^k)\big) \text{ if } \begin{cases} p \in P^{\hookrightarrow} \\ (r_1^k, \dots, r_n^k) \hookrightarrow (p, \lambda(p)) \end{cases} \end{cases}$$
- $I_s : T \rightarrow \mathcal{I}$ is defined $\forall t \in T$ by: $I_s(t) = I_{s_i}(t)$ **if** $t \in T_i$

As an example, $\mathcal{N} = \Big(\mathcal{N}_1 || \mathcal{N}_2 || \mathcal{N}_3\Big)\Big|_{(r_1, r_2, \bullet) \hookrightarrow (p, r)}$ is the parallel composition of the 3 TPNs, i.e., $\mathcal{N}_1$, $\mathcal{N}_2$ and $\mathcal{N}_3$, where the place $p_1 \in P_1$ such that $\lambda_1(p_1) = r_1$ and the place $p_2 \in P_2$ such that $\lambda_2(p_2) = r_2$ are merged. The name of the place obtained by this merging in $\mathcal{N}$ is $p \in P$ and its role is $\lambda(p) = r \in R$.

Property 1 (Associativity): The composition of TPNs with roles is associative in the following sense:

$$\Big(\mathcal{N}_1 || \mathcal{N}_2 || \mathcal{N}_3\Big)\Big|_{(r_1, r_2, r_3) \hookrightarrow (p, r)} = \Big(\Big(\mathcal{N}_1 || \mathcal{N}_2\Big)\Big|_{(r_1, r_2) \hookrightarrow (p_{12}, r_{12})} || \mathcal{N}_3\Big)\Big|_{(r_{12}, r_3) \hookrightarrow (p, r)}$$

$$= \Big(\mathcal{N}_1 || \big(\mathcal{N}_2 || \mathcal{N}_3\big)\Big|_{(r_2,r_3)\hookrightarrow(p_{23},r_{23})}\Big)\Big|_{(r_1,r_{23})\hookrightarrow(p,r)}$$

Property 2 (Commutativity): The composition of TPNs with roles is commutative:

$$\Big(\mathcal{N}_1 || \mathcal{N}_2\Big)\Bigg|_{\substack{(r_1^1, r_2^1) \hookrightarrow (p^1, r^1) \\ \ldots \\ (r_1^k, r_2^k) \hookrightarrow (p^k, r^k)}} = \Big(\mathcal{N}_2 || \mathcal{N}_1\Big)\Bigg|_{\substack{(r_2^1, r_1^1) \hookrightarrow (p^1, r^1) \\ \ldots \\ (r_2^k, r_1^k) \hookrightarrow (p^k, r^k)}}$$

V. Construction and illustration

The definitions presented above will help the formal construction of behavioral models in TPN. This construction will serve as a basis of the transformation process within SExPIsTools framework (Figure 1). To better understand the concepts involved in this construction, we must specify the major categories of concepts in RTEPML [2]. At the moment, three of them were selected from a behavioral point of view: concurrent resources (i.e., tasks, interruptions, alarms, etc.), interaction resources (i.e., semaphores, message queues, shared data, events, etc.) and routines (i.e., application code). For the sake of clarity, the construction has deliberately been splitted into four composition operations. The overall construction is a sequence of four operations.

A construction example in TPN is provided to illustrate the method. Figure 2 presents some TPN fragments instantiated with roles (in boxes), prepared for construction. Every operation details the fragments involved in the composition. The mergeable places are represented in double circle and those ready to be merged are connected by a hook-dotted arc with the number of the construction. The roles are assigned to the right above of places. The whole model is describing a monoprocessor application $Proc$ with two periodic tasks $Task_1$ and $Task_2$ sharing the same semaphore Sem_1.

a) Construction for each routine: The routines serve as executive body of concurrent resources. They consist of an ordered sequence of call services. The list of services considered in RTEPML is not exhaustive at the moment. The instructions described in TPN are: activation and termination of task, acquisition and release of semaphore and waiting, notification and inhibition of event.

Let n be the number of call services described following: $\{\mathcal{N}_{S1}, \mathcal{N}_{S2}, \ldots, \mathcal{N}_{Sn}\}$ such that $\forall i \in [1, n], \mathcal{N}_{S_i} = Ins(\mathcal{N}_S, S_i)$ with $\mathcal{N}_S$ the TPN describing a service. The routine construction then implies $n-1$ compositions, each one having m_j mergings of places with $j \in [1, n-1]$. The construction of a routine instance $\mathcal{N}_R$ is given by (1).

Illustration 1 (See Figure 2): In accordance with $\mathcal{N}_R$, $\forall l \in [1, 2]$, $\mathcal{N}_{Task_l Body}$ is built from TPNs $\{\mathcal{N}_{Get_l(Sem_1)}, \mathcal{N}_{Release_l(Sem_1)}, \mathcal{N}_{Terminate_l(Task_l)}\}$. This sequence describes in the order, an acquisition of Sem_1, a release of Sem_1 and a termination of $Task_l$.

b) Construction for each entry point of a concurrent resource: Each resource points to a routine described by $\mathcal{N}_R$ previously formed. Only one operation composes $\mathcal{N}_R$ with $\mathcal{N}_{C\lambda} = Ins(\mathcal{N}_C, C_\lambda)$. $\mathcal{N}_C$ is the TPN describing a concurrent resource. The construction of a concurrent resource instance with its executive body $\mathcal{N}_{CR}$ is given by (2) for m mergings.

Illustration 2 (See Figure 2): In accordance with $\mathcal{N}_{CR}$, $\forall l \in [1, 2]$, $\mathcal{N}_{Task_l_withBody}$ is built composing $\mathcal{N}_{Task_l}$ with its entry point $\mathcal{N}_{Task_l Body}$.

c) Construction for concurrent resources: At this stage, concurrent resources must be attached together with the aim of being scheduled by the same processor.

Let q_C be the number of concurrent resources with their composed executive bodies such that $\forall i_C \in [1, q_C]$, each resource is described by $\mathcal{N}_{CR_{i_C}}$ in accordance with $\mathcal{N}_{CR}$ previously formed. The construction then implies $q_C - 1$ compositions, each one having m_{j_C} mergings with $j_C \in [1, q_C - 1]$. The construction of $\mathcal{N}_W$ is given by (3).

Illustration 3 (See Figure 2): In accordance with $\mathcal{N}_W$, $\mathcal{N}_{withoutProc}$ is firstly composed of $\mathcal{N}_{Task_1_withBody}$ and $\mathcal{N}_{Task_2_withBody}$.

d) Global construction with processor and interaction resources: Note that the processor is also a shared resource. It will therefore be considered as an interaction resource.

Let q_I be the number of interaction resources considered such that $\forall i_I \in [1, q_I]$, each resource is described by $\mathcal{N}_{I_{i_I}} = Ins(\mathcal{N}_I, I_{i_I})$ with $\mathcal{N}_I$ the TPN describing an interaction resource. Each interaction resource is composed with $\mathcal{N}_W$ previously formed. The global construction then implies q_I compositions, each one having m_{j_I} mergings with $j_I \in [1, q_I]$. The global composition $\mathcal{N}_G$ is given by (4).

Illustration 4 (See Figure 2): In accordance with $\mathcal{N}_G$, $\mathcal{N}_{DeployedApplication}$ is finalized by composing $\mathcal{N}_{withoutProc}$, $\mathcal{N}_{Sem_1}$ and $\mathcal{N}_{Proc}$.

VI. Benefits and Limits

The use of TPNs with roles and the composition based on roles has allowed to detect several errors within the SExPIsTools transformation process. Those errors were bad transformation rules between concepts and roles, bad descriptions of the behavioral fragments.

That has also clarified the chaining of the transformation rules. As a result, a part of the transformation prototype has been rewritten. This approach has increased the confidence in SExPIsTools framework and its generated formal models.

Although SExPIsTools can consider several RTOSs, we have only defined fragments for OSEK/VDX [9] in TPN. Moreover, some complex real-time mechanisms, such as priority ceiling protocol or special queues of message box show the limits of the expressiveness of TPNs. For this reason, we could not model those mechanisms.

VII. Conclusion

An approach has been presented to build a formal model of RTESs taking into account a RTOS description. A new definition has extended the modeling in TPN to compose fragments with roles. The formalized composition will be used as a basis of the transformation process. The process implementation in SExPIsTools is in progress. The framework integrates a modeling language called RTEPML designed to describe the behavior of RTOSs. During the process running, only the description of the target execution platform is considered.

The main idea of this process is to maintain a genericity of implementation. Composition rules introduced in this paper are independant of any RTOSs thanks to role notion. This notion is an essential point of our strategy and brings an advantage in relation to other existing approaches. Future prospects are scheduled in order to take into account other RTOS descriptions. Another important point is the consideration of more

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

$$\mathcal{N}_R = \left(\left(\left(\mathcal{N}_{S_1}||\mathcal{N}_{S_2}\right)\Big|_{\begin{array}{l}(end_S_1, start_S_2) \hookrightarrow (S_{S_1 \to S_2}, \perp)\\ (r^2_{S_1}, r^2_{S_2}) \hookrightarrow (p^2_{S_2}, r^2_{S_2})\\ \dots\\ (r^{m_1}_{S_1}, r^{m_1}_{S_2}) \hookrightarrow (p^{m_1}_{S_2}, r^{m_1}_{S_2})\end{array}} ||\mathcal{N}_{S_3}\right)\Big|_{\begin{array}{l}(end_S_2, start_S_3) \hookrightarrow (S_{S_1 S_2 \to S_3}, \perp)\\ (r^2_{S_1 S_2}, r^2_{S_3}) \hookrightarrow (p^2_{S_3}, r^2_{S_3})\\ \dots\\ (r^{m_2}_{S_1 S_2}, r^{m_2}_{S_3}) \hookrightarrow (p^{m_2}_{S_3}, r^{m_2}_{S_3})\end{array}} \dots ||\mathcal{N}_{S_n}\right)\Big|_{\begin{array}{l}(end_S_{n-1}, start_S_n) \hookrightarrow (S_{S_1 S_2 \dots S_{n-1} \to S_n}, \perp)\\ (r^2_{S_1 S_2 \dots S_{n-1}}, r^2_{S_n}) \hookrightarrow (p^2_{S_n}, r^2_{S_n})\\ \dots\\ (r^{m_{n-1}}_{S_1 S_2 \dots S_{n-1}}, r^{m_{n-1}}_{S_n}) \hookrightarrow (p^{m_{n-1}}_{S_n}, r^{m_{n-1}}_{S_n})\end{array}} \quad (1)$$

with $\forall k \in [1, m_j]$ and $n \geq 2$ if $k \geq 2$ then $r^k_{S_1 \dots S_j} = r^k_{S_{j+1}}$

$$\mathcal{N}_{CR} = \left(\mathcal{N}_{C_\lambda}||\mathcal{N}_R\right)\Big|_{\begin{array}{l}(start_C_\lambda, start_S_1) \hookrightarrow (S, \perp)\\ (end_C_\lambda, end_S_n) \hookrightarrow (E, \perp)\\ (r^3_{C_\lambda}, r^3_R) \hookrightarrow (p^3_R, r^3_R)\\ \dots\\ (r^m_{C_\lambda}, r^m_R) \hookrightarrow (p^m_R, r^m_R)\end{array}} \quad (2)$$

with $\forall k \in [1, m]$ if $k \geq 3$ then $r^k_{C_\lambda} = r^k_R$

$$\mathcal{N}_W = \left(\left(\mathcal{N}_{CR_1}||\mathcal{N}_{CR_2}\right)\Big|_{\begin{array}{l}(processor_CR_1, processor_CR_2) \hookrightarrow (P_{CR_1 \to CR_2}, processor_Proc)\\ (r^2_{CR_1}, r^2_{CR_2}) \hookrightarrow (p^2_{CR_2}, r^2_{CR_2})\\ \dots\\ (r^{m_1}_{CR_1}, r^{m_1}_{CR_2}) \hookrightarrow (p^{m_1}_{CR_2}, r^{m_1}_{CR_2})\end{array}} \dots ||\mathcal{N}_{CR_{q_C}}\right)\Big|_{\begin{array}{l}(processor_CR_{q_C-1}, processor_CR_{q_C}) \hookrightarrow (P_{CR_1 \dots CR_{q_C-1} \to CR_{q_C}}, processor_PROC)\\ (r^2_{CR_1 \dots CR_{q_C-1}}, r^2_{CR_{q_C}}) \hookrightarrow (p^2_{CR_{q_C}}, r^2_{CR_{q_C}})\\ \dots\\ (r^{m_{q_C-1}}_{CR_1 \dots CR_{q_C-1}}, r^{m_{q_C-1}}_{CR_{q_C}}) \hookrightarrow (p^{m_{q_C-1}}_{CR_{q_C}}, r^{m_{q_C-1}}_{CR_{q_C}})\end{array}} \quad (3)$$

with $\forall k_C \in [1, m_{j_C}]$ and $q_C \geq 2$ if $k_C \geq 2$ then $r^{k_C}_{CR_1 \dots CR_{j_C}} = r^{k_C}_{CR_{j_C+1}}$

$$\mathcal{N}_G = \left(\left(\mathcal{N}_W||\mathcal{N}_{I_1}\right)\Big|_{\begin{array}{l}(r^1_P, r^1_{I_1}) \hookrightarrow (p^1_{I_1}, r^1_{I_1})\\ \dots\\ (r^{m_1}_P, r^{m_1}_{I_1}) \hookrightarrow (p^{m_1}_{I_1}, r^{m_1}_{I_1})\end{array}} \dots ||\mathcal{N}_{I_{q_I}}\right)\Big|_{\begin{array}{l}(r^1_{PI_1 \dots I_{q_I-1}}, r^1_{I_{q_I}}) \hookrightarrow (p^1_{I_{q_I}}, r^1_{I_{q_I}})\\ \dots\\ (r^{m_{q_I}}_{PI_1 \dots I_{q_I-1}}, r^{m_{q_I}}_{I_{q_I}}) \hookrightarrow (p^{m_{q_I}}_{I_{q_I}}, r^{m_{q_I}}_{I_{q_I}})\end{array}} \quad (4)$$

with $\forall k_I \in [1, m_{j_I}]$ and $q_I \geq 1$, $r^{k_I}_{PI_{j_I-1}} = r^{k_I}_{I_{j_I}}$

complex RTOSs mechanisms. The use of high-level Petri Nets such as Scheduling TPNs [5] is also planned.

Finally, a more long-term goal is planned to check the correctness of the transformation. A formal comparison between an application model projected on a more abstract platform and a deployed application model generated by SExPIsTools could allow this verification.

REFERENCES

[1] M. Boyer and O.H. Roux, "On the compared expressiveness of arc, place and transition time Petri nets," Fundamenta Informaticae, August. 2008, pp. 88(3):225-249.

[2] M. Brun and J. Delatour, "Contribution on the software execution platform integration during an application deployment process," First Topcased Day, Toulouse, February. 2011.

[3] W. El Hajj Chehade, A. Radermacher, S. Gérard, and F. Terrier, "Detailed Real-Time Software Platform Modeling," Software Engineering Conference (APSEC), 17th Asia Pacific, November. 2010, pp. 108-117.

[4] C. Lelionnais, M. Brun, J. Delatour, O.H. Roux, and C.Seidner, "Formal Behavioral Modeling of Real-Time Operating Systems," ICEIS(2) - Proceedings of the 14th International Conference on Enterprise Information Systems (Special Session on Model Driven Development for Information Systems: Techniques, Tools, and Methodologies - MDDIS 2012), Wroclaw, Poland: June. 2012, pp. 407-414.

[5] D. Lime and O.H. Roux, "Formal verification of real-time systems with preemptive scheduling," Journal of Real-Time Systems, Springer, February. 2009, pp. 41(2):118-151.

[6] M. Merlin, "A study of the recoverability of computing systems," PhD dissertation, Univ. of California, Department of Information ans Computer Science, Irvine, 1974.

[7] J. Miller and J. Mukerji, "Model Driven Architecture (MDA) Guide, version 1.0.1.," Technical report, Object Management Group, June. 2003.

[8] Object Management Group (OMG), "UML Profile for Modeling and Analysis of Real Time and Embbeded systems (MARTE), version 1.1.," Technical report, June. 2011.

[9] OSEK/VDX Group, "OSEK/VDX Operating System Specification, version 2.2.3.," Technical report, February. 2005.

[10] F. Peres, B. Berthomieu, and F. Vernadat, "On the composition of Time Petri Nets," Discrete Event Dynamic Systems, September. 2011, pp. 21(3):395-424.

[11] F. Taïani, M. Paludetto, and J. Delatour, "Composing real-time objects: a case for Petri nets and Girard's linear logic," Object-Oriented Real-Time Distributed Computing, ISORC-2001. Proceedings. Fourth IEEE International Symposium on, May. 2001, pp. 298-305.

[12] F. Thomas, S. Gérard, J. Delatour, and F. Terrier, "Software Real-Time Resource Modeling," Embedded Systems Specification and Design Languages, Lecture Notes in Electrical Engineering, Springer Netherlands, 2008, pp. 10:169-182.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

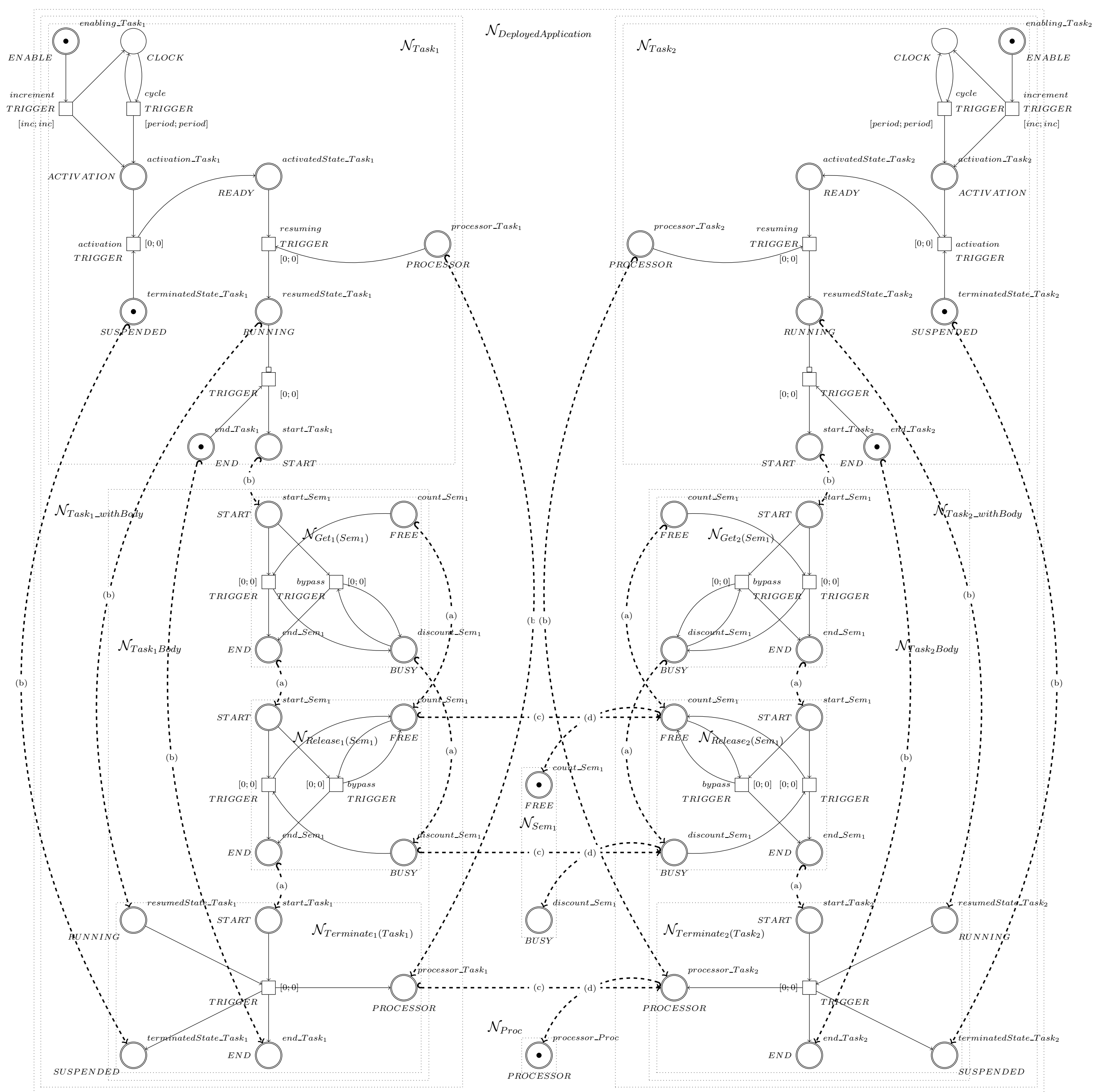


Fig. 2: Deployed application of semaphore sharing composed in TPN

Preliminary Test Suite Reduction

Vitaly Kozyura and Sebastian Wieczorek
SAP AG
Darmstadt, Germany
v.kozyura;sebastian.wieczorek@sap.com

***Abstract*—Test suite reduction is an activity which reduces test suites while maintaining their coverage properties. This problem is equivalent to the set covering problem and therefore NP-complete. Many strategies for solving the problem are known. They are usually applied to minimizing the number of action calls within a given test suite for a certain coverage goal. While some algorithms like branch and bound compute an exact minimal solution, other algorithms like the greedy approach compute an approximation for the minimal set of actions. In this work, we deal with the problem of efficient test suite reduction in industrial practice. For this purpose, we introduce the concept of preliminary test suite reduction. Its aim is to reduce redundancy in test suites before starting the actual reduction. In the paper, we further describe experimental results that give implication on how the proposed technique can reduce the runtime of test suite reduction in the industrial practice.**

Keywords—MBT; test suite reduction; industrial case study.

I. Introduction

Automatically generating tests suites from formal specifications as advertised by model-based testing (MBT) is regarded as a potential innovation leap in industrial software quality assurance. Most MBT approaches are running in two phases. In the first phase, vast amounts of test cases are generated for an inserted model until coverage of model entities is achieved. In the second phase, a subset of these test cases is selected with the aim to preserve the targeted coverage and therefore the assumed fault-uncovering capabilities [1]. This activity is called test suite reduction.

Fig. 1 represents a test model, that will be used as a running example in order to illustrate the reduction techniques. The test model is given in form of a finite state machine. The states are depicted as circles and the actions as named arrows. $q0$ is a start state and the state with two nested circles is an end state. A valid test is a sequence of actions starting in $q0$ and ending in the end state. As the coverage criteria we choose to cover all action names. Please note that in practice the execution of actions may be constrained by input and system data and may have additional side effects on data apart from state changes [2]. How side effects may be handled later on during test case execution is sketched in Section V-A.

Assuming that a model checking technique (breadth-first search) is used for the test generation in phase one,

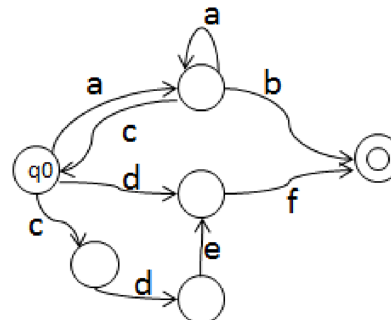


Fig. 1. Running Example.

the following test suite may be obtained:

$$[ab, df, aab, aaab, acab, acdf, cdef].$$

It can be noted that the derived test suite covers all action names, but it is not the minimal test suite (measured by the overall number of actions). How this test suite can be reduced, while preserving action name coverage will be discussed in the consequent sections.

The problem of test suite reduction is largely discussed in the literature. There are papers, where the general test suite reduction activity is described [3], [4]. Further work on how to apply 0/1-Integer linear programming to the test suite reduction problem [5] or how to improve the Greedy heuristics [6], [7], [8] can be found. In [9], [10] there are approaches using multi-objective optimization functions, whereas in [11] an approach based on genetic algorithms is introduced. Some empirical results for test suite reductions have been reported in [12].

This paper is motivated by the problem of *efficient* test suite reduction. We introduce a preliminary test suite reduction technique, aiming to reduce the runtime of the test suite reduction procedure and investigate its applicability to test suite reduction in the industrial practice of MBT.

First, we provide the industrial context for test suite reduction (Section II), then we describe the test suite reduction problem in detail (Section III). In this paper, we consider two existing approaches to the test suite reduction problem: approximative technique (described on the example of a greedy algorithm [13]) and the technique searching for an exact solution (described on the example of a branch and bound algorithm [14]).

In Section IV, we define the concept of *preliminary test suite reduction.* In essence, the goal of applying preliminary reduction is to make the overall activity of test suite

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

reduction more efficient. In Section V we provide a few experimental results in order to illustrate the proposed technique and also to discuss its applicability for practical test suite reduction. Section VI concludes the paper.

II. Industrial Context

In the software industry, model-based test automation is one of the most promising approaches for increasing the efficiency of testing. Various commercial vendors emerged that offer tools and consulting, maintain user groups, and organize industrial conferences. Although the various competing commercial tools utilize alternative test generation concepts like model checking, theorem proving, or random walks, the overall process of producing test suites is similar. It consists of two phases:

1) Test suite generation - deriving test cases from the model, usually until test coverage is reached.
2) Test suite reduction - calculating a subset of test cases that maintains test coverage.

Especially in industrial settings, the second phase is indispensable, because of the significant manual effort associated with test case concretization [15]. This transformation from abstract test cases to executable test scripts usually follows the keyword-driven testing principles. Keyword-driven testing uses keywords in the test cases, in addition to data. Each keyword corresponds to a fragment of a test script (the adapter code), which allows the test execution tool to translate a sequence of keywords and data values into executable tests [1].

Generated test suites usually contain a large number of redundant test cases, that would unnecessarily increase the concretization effort. For example, the initially generated test suite given in Section I contains 7 test cases and 23 actions. However, various subsets of the given test suite exist that are preserving the defined coverage of all action names. As to be shown in Section III, the optimal solution only contains 2 test cases and 6 actions.

The most common reasons for redundancy in test suites are the presence of loops in the test model as well as multiple occurrences of equal action names. The given example contains both. Further, some test generation approaches deliberately continue to create test cases despite the fact that the computed test suite already meets the coverage criteria. Often, this enables better reduction results, as it increases the variety of test cases.

In industrial practice, various additional sources of redundancy may exist that are not connected to the model structure or test generator. For example, it is often the case that after initial thorough testing, test suites with reduced coverage requirements are created to lower the execution runtime of regression tests. In order to avoid the effort of test re-generation and especially the potential test concretization of additional test cases, usually the already generated and used test suite is reduced again. Also, the merging of generated test suites with manually designed or legacy test cases often occurs in practice.

III. Test Suite Reduction

As described, for a provided model the obtained test suite can contain a very large number of test cases. Aim of the test suite reduction is to select a subset, which preserves the targeted coverage and therefore the assumed fault-uncovering capabilities. This activity can be formulated like follows [7]:

Given: A test suite TS, a set of test case requirements $r_1, r_2, ..., r_n$ that must be satisfied to provide the desired testing coverage of the program, and subsets of TS, $T_1, T_2, ..., T_n$, one associated with each of the r_i's such that any one of the test cases tc_j belonging to T_i can be used to test r_i.

Problem: Find a representative set of test cases from TS that satisfies all of the r_i's.

The test suite reduction problem can be considered as a hitting set problem — the problem of finding the hitting set having minimum cardinality, which is equivalent to the set cover problem and is known to be NP-complete [16]. The standard way of solving the hitting set problem is a restatement into a 0/1-Integer linear program. Afterwards, this can either be exactly solved by using a technique like branch and bound algorithm or approximately, for example by applying different variations of Greedy heuristics [17], [13]. In the following, both approaches are described.

A. Greedy Algorithm

We use a classical implementation of the Greedy algorithm which has already been known for some time. Even though the Greedy algorithm computes an approximation, [13] showed that the result cannot become arbitrarily bad. In fact the upper bound for the performance guarantee only depends on the number of requirements.

The algorithm iteratively constructs subsets $\mathrm{TS}_i \subseteq \mathrm{TS}$, which will produce a complete test suite after termination of the algorithm. Until all requirements are met, the algorithm does the following: It computes the set of all test cases, for which the number of additional action calls is maximal. Then, it picks one of these (tc) at random. This test case is afterwards added to $\mathrm{TS}_{i+1} = \mathrm{TS}_i \cup \{tc\}$ and the requirements are updated appropriately.

The algorithm has a linear time complexity $O(|\mathrm{TS}|)$ with respect to the size of a test suite $|\mathrm{TS}|$ to be optimized.

Using a greedy algorithm on the initially generated test suite of the example given in Section I, the following reduced test suite can be obtained, containing 3 test cases and 10 actions:

$$[acdf, ab, cdef].$$

B. Branch and Bound Algorithm

In order to find an exact solution to the test reduction problem, we use the branch and bound variation (Balas-algorithm) described in [14], which allows one to compute an optimal result.

The algorithm identifies all possible subsets of $\mathrm{TS} = \{\mathrm{tc}_1, \ldots, \mathrm{tc}_m\}$ with arrays $(n_1, \ldots, n_m)$. Here $n_i = 1$

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

means, that tc_i is part of the subset, while $n_i = 0$ means, that it is not. To check these arrays systematically, they are organized as a binary tree. At the root node, no decisions have been made, whereas any node on level i represents a certain choice of the first i bits. The node $(n_1, \ldots, n_i)$ is identified with the test suite $\{\text{tc}_j : n_j = 1\}$. During the so-called *pruning*, it is checked for each constructed node if this node can be safely discarded from the tree.

In worst case, the algorithm has an exponential run time with respect to the size of a test suite $|\text{TS}|$ to be optimized.

In the case of using a branch and bound algorithm on the initially generated test suite of the example given in Section I, the following reduced test suite is obtained, containing 2 test cases and 6 actions:

$$[ab, cdef].$$

IV. Preliminary Reduction

As described, test suite reduction is necessary because initially generated test suites usually contain far more test cases than necessary to achieve certain coverage goals. On the other side, also test suite reduction itself may be a costly operation. As discussed in Section III, the runtime of test suite reductions can vary from linear to exponential depending on how exact the solution should be. As test engineers expect fast feedback from test automation tools, e.g. in order to not loose their focus, each runtime improvement for test suite reduction retaining the exactness of the solution can be very valuable from the practical point of view.

In this section, we would like to introduce the concept of preliminary removing redundancy in an initial test suite in order to reduce the runtime of the actual reduction procedure. The proposed preliminary reduction is applied before the actual optimization procedure starts. We define prelimiary redundancy as follows.

Definition: Given a test suite TS, we say that the test case tc is *redundant* if there exists another test case tc' with $|\text{tc}'| \leq |\text{tc}|$ and for each requirement r_i it holds $r_i(\text{tc}) \Rightarrow r_i(\text{tc}')$.

Following algorithm can be used in order to delete the redundant test cases from the test suite before starting the actual test suite reduction.

Algorithm: Our preliminary reduction procedure compares the test cases from TS with each other and deletes all redundant test cases. This results in a time complexity of $O(|\text{TS}|^2)$.

Applying the above definition to the example from Section I, a given test case is redundant if another test case with equal or less actions exists that covers at least the same action names. For instance, the test $aaab$ is redundant because $|ab| < |aaab|$ while both cover the same set of action names. The test suite obtained after applying the preliminary reduction to the example consequently is

$$[ab, df, acab, acdf, cdef].$$

As mentioned before, the greedy algorithm has a linear run time complexity, whereas the branch and bound algorithm has an exponential run time complexity with respect to the size of a test suite $|\text{TS}|$ to be optimized. Based on this information, the decision should be to always use preliminary reduction when constructing an exact minimal set of test cases and to never use it when constructing an approximative solution. However, in practice there are two factors, which can influence this decision:

- The typical regions of the test suite sizes and the grades of the run time complexity curves in these regions for each reduction algorithm.
- The typical rate of redundancy in the considered test suites.

Combining this two factors can lead to the situations where either it would not be reasonable to use preliminary reduction at all (also when constructing an exact minimal set of test cases) or where it would be reasonable to use it even when constructing an approximative solution to the test suite reduction problem.

After being able to construct examples for all these situations, we decided to check the applicability of preliminary optimization for both exact and approximative approaches to the test suite reduction problem in the industrial practice. The purpose is to derive a practically applicable guidance on the basis of the experiments with typical test suites from the industrial context.

V. Experimental Results

In this section, we present experimental results illustrating the applicability of preliminary reduction technique for optimizing the test suite reduction procedure. We start with a description of the industrial testing setup: how test models are constructed and how the tests are generated. Then, we provide the experimental results and discuss the usability of preliminary reduction in the industrial practice. Finally, possible threats to validity are described.

A. Test Setup

The testing approach we consider in this paper leverages a keyword-driven testing framework, as described in [18]. A model editor is implemented on top of the framework in order to facilitate an automated test generation. Test models are represented by transition state machines enhanced with data flow and global data.

As an input for the experiments, we have collected a number of test models, which were designed on the basis of industrial case studies. These test models were created for system testing. In practice, system testing is based on high-level usage scenarios and business requirements that have been defined by business analysts or customers. UI-based testing is most appropriate to carry out the tests, as the system should be validated as a whole and only using access points that are available to the prospect user.

Keyword-based testing for UI is mostly done by utilizing capture/replay functionality, which is provided by

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

standard test automation tools. These tools are monitoring user interactions on the interface that can reproduce the execution of the recorded sequence of events. These captured scripts commonly allow data flexibility by exchanging the concrete values (used during capturing) with variables that can be initialized independently.

Further, the recorded scripts can be combined in so-called scenarios. A scenario is a sequence of recorded scripts working on a predefined set of data. Global variables are used in scenarios to organize the data flow. Their function is to store output values of a captured script and make it available as input for another. In Fig. 2 an example of the described data flow is given. The value of the local output variable *A.out* of *Script A* is written to the global variable *X* and later mapped to the local input variable *B.in* of *Script B*.

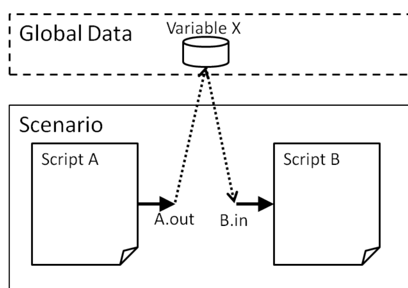


Fig. 2. Data flow in a scenario.

In order to allow a calculation of an exact minimal test suite and further to get realistic statements for the context of our work, most of the chosen samples are small- or intermediate-sized (I-IX). We also included one interesting border case example (NA), which is too large to be optimized with algorithms of the branch and bound type.

All computations were performed on an AMD Opteron (tm) Quad Core with 2.60 GHz and 32 Gigabytes of RAM.

B. Selected Results

In Table I and Table II, we present selected results of our experiments demonstrating the applicability of the preliminary reduction for an approximative and an exact construction of the reduced test suite. In Table I, we compare approximating greedy algorithm with and without preliminary reduction and in Table II we do the comparison for the branch and bound algorithm. The tables have the following columns: number of an example (Example), size of a test suite to be optimized (|TS|), size of a test suite obtained after preliminary reduction (|TS|(PR)) , and the run time for the each algorithm with and without preliminary reduction (Time(PR) / Time). The time is measured in seconds.

C. Discussion

As it can be seen in Table I and Table II, the preliminary reduction does not improve the approximative greedy algorithm, except for one border case (NA), where the optimal test suite (containing 3 test cases) is already obtained from 3160 test cases after the preliminary reduction. In contrast, branch and bound algorithms, which have exponential complexity, perform already better for the size of the fourth example (IV) if using the preliminary reduction.

TABLE I. GREEDY ALGORITHM WITH AND WITHOUT PRELIMINARY REDUCTION.

Example	|TS|	|TS|(PR)	Time	Time(PR)
I	15	14	0,06	0,08
II	32	32	0,07	0.12
III	41	41	0.08	0.14
IV	45	24	0,09	0,16
V	120	120	0,10	1,05
VI	132	111	0,62	2,12
VII	289	203	1,36	7,63
VIII	512	336	3,67	20,13
IX	625	402	4,83	44, 37
NA	3160	3	32,13	3,57

TABLE II. BRANCH AND BOUND ALGORITHM WITH AND WITHOUT PRELIMINARY REDUCTION.

Example	|TS|	|TS|(PR)	Time	Time(PR)
I	15	14	0,06	0,07
II	32	32	3,66	3,67
III	41	41	9,05	9,06
IV	45	24	0,13	0,11
V	120	120	77,49	77,50
VI	132	111	268,78	155,14
VII	289	203	511,02	403,71
VIII	512	336	1353,37	810,81
IX	625	402	1733,16	886,95
NA	3160	3	-	3,59

This means that in the industrial practice the most challenging test suites are located in the size region, where there is enough redundancy to make preliminary reduction an efficient technique.

In order to better understand the presented results, we also provide scatter diagrams representing the ratios for speedup or slowdown in the run time for the examples I-IX (Fig. 3). For each diagram on the x-axis, there are numbers of tests in the test suites and on the y-axis there are the corresponding ratios between run times with and without preliminary reduction (*Time(PR)/Time*). The thick horizontal lines define the areas where *Time(PR)/Time = 1*, i.e., the preliminary reduction brings neither advantages nor disadvantages from the run time perspective. The skew lines (trend lines) represent the correlations between the test suite size and the runtime ratio described above.

From the presented diagrams, it can be seen that in the case of approximative test suite reduction it is not only unsuitable to use preliminary reduction, but the drawback is increasing with the growing size of test suites. Otherwise, for the branch and bound algorithm the trend line shows that the value of using preliminary reduction grows with the growing size of test suites.

It can be seen that it is not always reasonable to apply it for small test suites, but the larger the test suites get the more beneficial it is to apply the preliminary reduction technique. From the practical point of view, the test suites of medium or large size are crucial with respect to run time, whereas for small test suites the possible slowdown is usually not critical.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

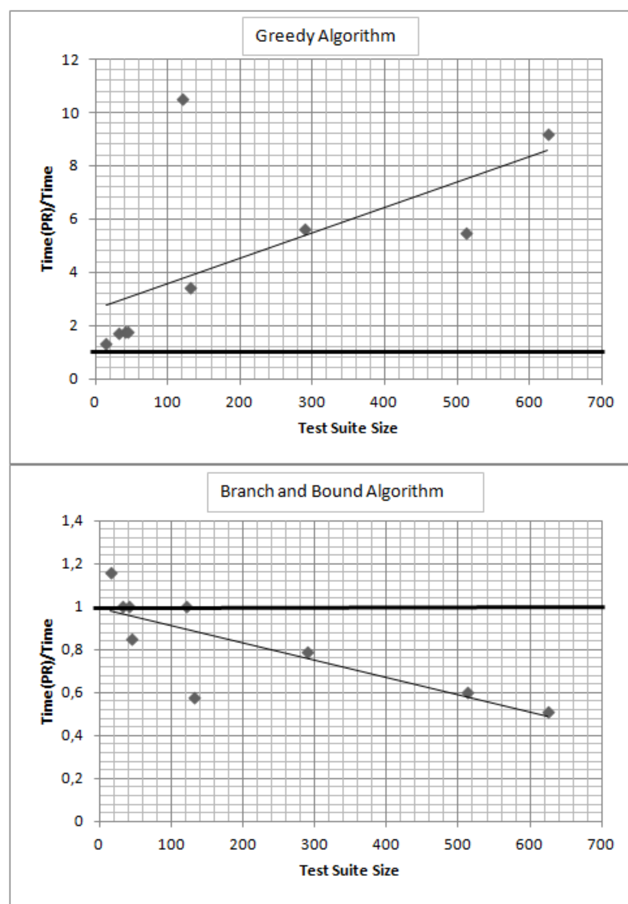


Fig. 3. Relative runtime diagrams.

Therefore, as a bottom line we deducted the following from the experiments:

For the industrial practice, it is recommended to use the preliminary reduction each time a (near) optimal solution for the test suite reduction problem is computed.

D. Threats to Validity

We realize that a number of experiments in the area of UI-based MBT cannot serve as a proof of applicability for the whole industrial area of MBT. However, we believe that the presented results can be generalized to the practical testing of high-level usage scenarios, where UI-based testing is the most commonly used approach.

VI. Conclusion

In this paper, we introduced the concept of preliminary test suite reduction and studied how eliminating redundant test cases can accelerate the test suite reduction algorithms.

We further described the industrial context of MBT and provided a collection of common reasons for the existence of the redundancy in test suites. The applicability of preliminary test suite reduction for the industrial practice of MBT is shown, based on a number of experiments from UI-based testing, which is a common way of testing the high-level scenarios.

We applied the preliminary optimization technique to two classical solutions of the test suite reduction problem, namely the branch and bound algorithm, which computes an exact solution in exponential time, and the greedy heuristic, which yields the best approximation possible in polynomial time. In the paper, we presented selected experimental results, which have shown that the approach pays off for branch and bound algorithms, but is rather inefficient for greedy algorithms.

From our industrial experience in MBT, we know that redundancy in the test suite is a common issue which affects test efficiency on various levels. Therefore, it can be an important aspect in practice to apply preliminary optimization in case a (near) optimal solution for the test suite reduction problem should be computed.

References

[1] M. Utting and B. Legeard, *Practical model-based testing, a tools approach.* Morgan Kaufmann, 2007.

[2] S. Wieczorek, A. Stefanescu, and I. Schieferdecker, "Test data provision for ERP systems," in *Proc. of Int. Conf. on Software Testing (ICST'08).* IEEE Computer Society, 2008, pp. 396–403.

[3] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, pp. 135–141, 1996.

[4] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *In Proc. Twelfth Int. Conf. Testing Computer Software*, 1995, pp. 111–123.

[5] H. S. Wang, S. R. Hsu, and J. C. Lin, "A generalized optimal path-selection model for structural program testing," *Journal of Systems and Software*, vol. 10, no. 1, pp. 55 – 63, 1989.

[6] R. Gupta and M. L. Soffa, "Compile-time techniques for improving scalar access performance in parallel memories," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 138–148, April 1991.

[7] M. J. Harrold, C. Unwersity, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, pp. 270–285, 1993.

[8] T. Y. Chen and M. F. Lau, "A new heuristic for test suite reduction," vol. 40, no. 5-6, pp. 347–354+, 1998.

[9] J. Black, E. Melachrinoudisl, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 106–115.

[10] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 140–150.

[11] N. Mansour and K. El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing," *Journal of Software Maintenance*, vol. 11, pp. 19–34, January 1999.

[12] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Journal of Software Testing, Verification, and Reliability*, vol. 12, pp. 219–249, 2002.

[13] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, pp. 233–235, 1979.

[14] J.W. Chinneck, "Practical Optimization: A Gentle Introduction," http://www.sce.carleton.ca/faculty/chinneck/po.html, 2003, chapter 13.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

[15] S. Wieczorek, A. Stefanescu, and I. Schieferdecker, "Model-based integration testing of enterprise services," in *Proc. of Testing: Academic & Industrial Conference - Practice and research techniques (TAICPART'09)*. IEEE Computer Society, 2009, pp. 56–60.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[17] V. V. Vazirani, *Approximation algorithms.* Springer, 2001.

[18] S. Wieczorek and A. Stefanescu, "Improving Testing of Enterprise Systems by Model-Based Testing on Graphical User Interfaces," in *2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems.* IEEE, 2010, pp. 352–357.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

Performance Characterization of TAS-MRAM Architectures in Presence of Capacitive Defects

João Azevedo, Arnaud Virazel, Yuenqing Cheng, Alberto Bosio, Lugi Dilillo, Patrick Girard, Aida Todri
LIRMM – University of Montpellier 2 / CNRS
Montpellier, France
e-mail: {azevedo, Virazel, cheng, bosio, dilillo, girard, todri}@lirmm.fr

Jérémy Alvarez-Hérault
CROCUS Technology
Grenoble, France
e-mail: jherault@crocus-technology.com

***Abstract*—Magnetic Random Access Memory (MRAM) is an emerging memory technology. Among existing MRAM technologies, Thermally Assisted Switching (TAS) MRAM technology offers several advantages such as selectivity, single magnetic field and high integration density. In this paper, we analyze the impact of capacitive defects on the TAS-MRAM performance. Electrical simulations were performed on a 16-words TAS-MRAM architecture enabling any sequences of read/write operations. Results show that writing operations may be affected by these defects. Especially, we demonstrate that some capacitive defects may have a local (single cell) impact on the functionality of TAS-MRAM while others, even if there is an effective coupling, do not change the functional operation. These results will be further used to develop effective test algorithms targeting faults related to actual defects that may affect TAS-MRAM architecture.**

Keywords–non-volatile memories; spintronics; TAS-MRAM; capacitive defects; fault modeling; test.

I. Introduction

Nowadays, Non-Volatile Memories (NVMs) are more and more integrated in consumer applications. Though widely used, Flash memories still have several drawbacks such as high supply voltage requirement, low speed and susceptibility to reliability issues due to high electric field for programming operations. On the other hand, Magnetic Random Access Memory (MRAM) is an emerging technology with high data processing speed, low power consumption and high integration density compared with Flash memories. Moreover, this memory technology is non-volatile with fair processing speed and reasonable power consumption when compared to Static RAMs (SRAMs). MRAM probably is the closest to an ideal "universal memory" and thus may be used as NVM as well as SRAM and DRAM according to the 2011 International Technology Roadmap for Semiconductors (ITRS) [1].

MRAMs have the potential to mitigate almost all Flash related issues but they are prone to defects as any other kind of memory. Only few papers on MRAM testing can be found in the literature, and target mainly Field Induced Magnetic Switching (FIMS) MRAM technologies. Su et al. [2] present the Write Disturbance Fault (WDF) model, a fault that affects data stored in Toggle MRAM cells due to the amount of magnetic field applied during write operations on neighboring cells. Su et al. [3] identified two new faults related to the magnetic junction behavior and called Multi-Victim Fault (MVF), in which a cluster of cells can easily change their magnetization state due to process variations, and Kink Fault (KF), in which the hysteresis loop shrinks because of its relation with cell shape, thus changing MTJ resistivity.

A thorough investigation and deep analysis must be done for testing MRAMs memories. In [4] and [5], resistive-open and resistive-bridge defect analyses are presented for TAS-MRAM architectures. These studies have revealed the importance of electrical analyses of defects that may impact the performance of TAS-MRAMs.

In this paper, we complete these studies by considering parasitic coupling, i.e., capacitive defects. Parasitic coupling between adjacent interconnect lines is a major limiting factor in deep-submicron ICs due to the injection of noise from switching lines to neighboring lines [6]. The trend of increasing the integration level of ICs has a negative impact on interconnect performance. The cross section is smaller in the scaling-down process increasing the line's resistance. In order to reduce resistance while maintaining high horizontal interconnect density, the aspect ratio is larger than "1" increasing the coupling capacitance. In addition, the effective capacitance increases as the spacing between lines decreases, which causes an increase in the delay related to the RC constant. Moreover, crosstalk between lines due to mutual capacitance and inductance becomes worse.

In this context, we fully characterize the impact of capacitive defects on the TAS-MRAM performance. Considered defects were selected based on the layout structure of the TAS-MRAM array. Simulations were performed in a TAS-MRAM architecture supporting any read/write sequences. Results show that capacitive defects have almost no impact on read operations. Conversely, write operations are affected by capacitive defects depending on the size and location. Such results will be helpful to define an efficient test algorithm to fully test TAS-MRAMs.

The rest of the paper is organized as follows. Section II provides the fundamentals and background on MRAM technologies. The proposed TAS-MRAM architecture is described in Section III. The capacitive defect analysis is provided in Section IV. Section V concludes the paper.

II. MRAM Technologies

MRAMs are Spintronic devices that store data in Magnetic Tunnel Junctions (MTJs). A basic MTJ device is usually composed of two FerroMagnetic (FM) layers separated by an insulating layer, as shown in Figure 1. One of the FM layers is pinned and acts as a reference layer. The other one is free and can be switched between, at least, two stable states. These

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

states are parallel or anti-parallel with respect to the reference layer. When the MTJ is in the parallel state, it offers the minimum resistance (*Rmin*) while the maximum resistance (*Rmax*) is obtained when anti-parallel. The difference between *Rmin* and *Rmax*, quantified by the Tunnel Magneto Resistance (TMR), is high enough to be sensed during the read operation.

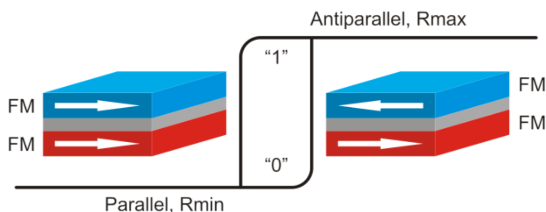


Figure 1. MTJ in parallel and antiparallel states

A read operation consists in determining the MTJ's magnetization state and can be performed by voltage or current sensing across the MTJ stack. A CMOS sense amplifier is used to retrieve the stored bit information. High TMR allows simple and stable sense amplifiers, improving reading accuracy [7].

Magnetization dynamics describes how the magnetization goes from one point of equilibrium to another one. This evolution of the magnetization in terms of time and space under a local effective field can be described by the Landau-Lifshitz-Gilbert equation (1):

$$\frac{\partial \vec{m}}{\partial t} = -\frac{\gamma}{1+\alpha^2}\vec{m}\times\overrightarrow{H_{eff}} - \frac{\gamma\alpha}{1+\alpha^2}\vec{m}\times\left(\vec{m}\times\overrightarrow{H_{eff}}\right) \quad (1)$$

where $\vec{m}$ is the unit vector along the magnetization of the free layer, γ is the gyromagnetic ratio, α is the Gilbert damping constant and H_{eff} is the effective magnetic field.

A write operation can be performed using magnetic fields or spin polarized current and depends on MRAM technologies: FIMS (Field Induced Magnetic Switching), Toggle Switching, TAS (Thermally Assisted Switching) and CIMS (Current Induced Magnetic Switching).

Thermally Assisted Switching™ is an alternative switching method for MRAMs. In the scheme proposed by Spintec [8] and industrialized by Crocus Technology, the MTJ is modified by inserting an Anti-FerroMagnetic (AFM) layer that pins the storage layer while below its blocking temperature (T_B) that can be calculated by (2).

$$T_B = \frac{KV}{k_B \ln\left(\tau_0 e^{\left(\frac{KV}{k_B T}\right)}\right)} \quad (2)$$

where K is the effective anisotropy constant, V is the device volume, k_B is the Boltzmann constant and τ_0 is the attempt time.

In AFM materials, the magnetic moments of atoms are aligned in a regular pattern, neighboring spins pointing in opposite directions. This organization vanishes above T_B and the material becomes paramagnetic. When MTJ's temperature rises above T_B, the storage layer is freed and can be reversed under the application of a small magnetic field provided by a single field-line. The magnetic field is maintained beyond the heating voltage pulse to ensure the correct pinning of the storage layer.

TAS approach offers several advantages compared to predecessors MRAM technologies. The selectivity problem is reduced since only heated MTJs are able to switch and all other MTJs hold their stable state as they remain below their blocking temperature. Although TAS-MRAM needs an additional heating current, this current is much smaller than the current used to generate the second magnetic field in FIMS-MRAM technology. The integration density is improved due to thermal stability and the need of only one field-line. Finally, as the free layer can be pinned to any stable state, multi level logic can be achieved [9]. TAS-MRAM is for the moment the most promising MRAM solution as it mitigates most drawbacks from its predecessors.

III. TAS-MRAM ARCHITECTURE

Figure 2 shows the TAS-MRAM architecture we have developed for our study. The organization is done in a square matrix that has 2^{MR} rows and 2^{NC} columns, for a total storage capacity of 2^{MR+NC} bits per page, where MR and NC are the numbers of bits used to specify the row and column address, respectively. In our case study, MR and NC are equal to 2 and the number of pages is 4; hence, the storage capacity is 64 bits (16 words of 4 bits). Each cell in the array is connected to one of the row-lines, called word-lines, and connected to one of the column-lines, called bit-lines. A particular set of MTJs can be accessed for a read or write operation by selecting its word-line and bit-line. There is only one field-line that connects all MTJs serially row by row passing through all pages in this architecture.

During a read operation, the read driver applies a small voltage that generates negligible heat to both the selected MTJ and a reference MTJ. The reference MTJ is halfway between the high and low resistance values. The resistance difference is then sensed to determine the stored data in the selected MTJ.

A write operation is performed as follows:

- Initially, the write driver applies a voltage to heat the selected MTJ above its T_B (about 150 °C).
- Next, the field-line driver applies a current to generate the data zero magnetic field. While the MTJ is cooled down below T_B, the magnetic field is maintained.
- Then, the field-line driver inverses the current direction and the MTJ is heated again to perform the write 1 operation, if needed. When the MTJ reaches room temperature, the writing procedure is accomplished.

This approach allows writing "logic 0" and "logic 1" in one cycle to different MTJs sharing the same field-line. Note that, a Write 1 operation (denoted W1) consists of applying both field-line current polarities (magnetic field for "data 0" and then magnetic field for "data 1"), while a Write 0 operation (denoted W0) consists of applying only one field-line current polarity (magnetic field for "data 0" only). These writing procedures are inspired by Flash programming procedures

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

where a write operation (write 1) is always preceded by an erase operation (write 0).

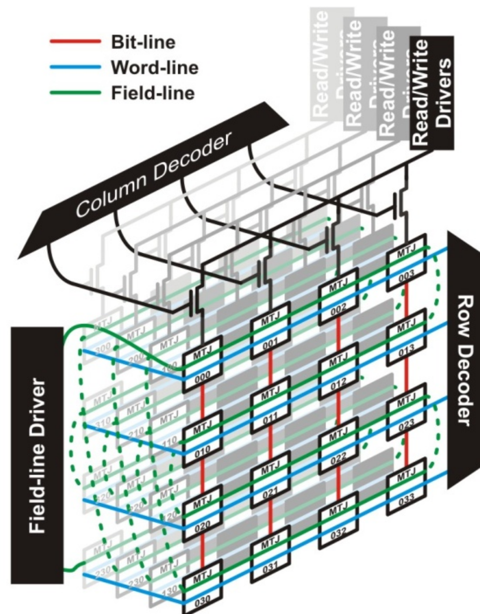


Figure 2. TAS-MRAM architecture

Electrical simulations were performed using the TAS-MTJ model developed by Spintec [9]. This model is based on the physical equations of the MTJ and is calibrated with respect to the targeted TAS-MRAM technology. Moreover, this model is compiled in C language and is compatible with the Spectre simulator of the standard Cadence design suite [10].

Table I summarizes simulated fault-free characteristics of $MTJ_{1,1,1}$ ($MTJ_{i,j,k}$ with i ⇒ page number, j ⇒ row number and k ⇒ column number) in the second page, second column and second line of the TAS-MRAM architecture shown in Figure 2. The first column gives the four possible operations R0, R1, W0 and W1. The next five columns provide all the MTJ's parameters:

- **V** – Voltage level at the MTJ interface.
- **I** – Current passing through the MTJ during read or write operations.
- **R** – Resistance of the MTJ.
- **T** – Temperature of the MTJ during operations.
- **M** – Magnetization state that is related to the angle between the two ferromagnetic layers. The parallel magnetization state is represented ideally by "1 ⇒ logic 0" and the anti-parallel magnetization state by "-1 ⇒ logic 1". The magnetization state is correlated to the resistivity of the MTJ.

Finally, the last column gives the sensing voltage (S) during read operation only. The two resistive states of the MTJ are 1.48kΩ for *Rmin* and 2.80kΩ for *Rmax* during read operation. In normal operation the sensing voltage (S) is around 165mV for *Rmin* and 254mV for *Rmax*. During write operations, the current that passes through the MTJ is high enough to heat its temperature above the blocking temperature, i.e., 193°C for W0 and 193/183°C for W1. The MTJ's resistivity is different during read and write operations even if the magnetization state is the same. This is due to the voltage applied to the MTJ as well as its operating temperature.

TABLE I. $MTJ_{1,1,1}$ CHARACTERISTICS UNDER READ/WRITE OPERATIONS

Operation	$MTJ_{1,1,1}$ parameters					S (mV)
	V (mV)	I (uA)	R (kΩ)	T (°C)	M	
R0	111.49	74.89	1.48	31.16	1	165.67
R1	202.35	72.11	2.80	34.26	-1	254.49
W0	745.32	606.59	1.22	193.18	1	n.a.
W1	745.32 863.09	606.59 542.63	1.22 1.59	193.18 183.75	-1	n.a.

Figures 3 and 4 show temperature profiles for W0 and W1 operations performed on a 16-bit fault-free TAS-MRAM memory page, respectively. Note that, MTJs are written row by row from $MTJ_{x,0,0}$ to $MTJ_{x,3,3}$. We observe that temperature rises twice during each write cycle in W1 operations and rises only once per cycle in W0 operations. This behavior is expected according to the writing scheme previously described.

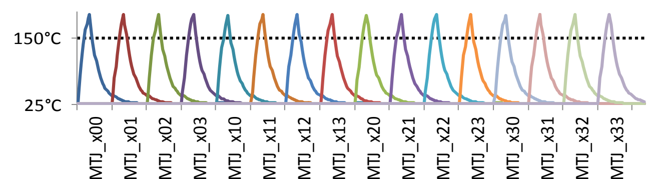


Figure 3. 16-bit W0 fault-free temperature profiles

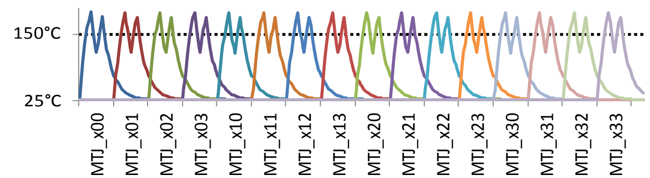


Figure 4. 16 bits W1 fault-free temperature profiles

IV. CAPACITIVE DEFECT INJECTION

The capacitive defect injection in TAS-MRAM architecture is depicted in Figure 5. Capacitive defects are inserted between interconnect lines based on the TAS-MRAM array layout as follows:

- **C1:** MTJ's bottom metal – Field-line
- **C2:** Field-line – Word-line
- **C3:** Word-line – Word-line
- **C4:** Word-line – MTJ's bottom metal

Note that all resistive parameters are not represented in Figure 5 but are all taken into account in models (lines, transistors, drivers) used for electrical simulations.

TAS-MRAM performance is affected by these capacitive defects in several ways. In the following sub-sections, we show a complete analysis of how these four capacitive defects impact the TAS-MRAM performance. Simulations were performed using the following write sequences:

Copyright (c) IARIA, 2013.

- 0W0: W0 operation performed on a MTJ that initially contains “logic 0”. There is no transition in this sequence.
- 1W0: W0 operation performed on a MTJ that initially contains “logic 1”. This sequence corresponds to a falling transition.
- 0W1: W1 operation performed on a MTJ that initially contains “logic 0”. This sequence corresponds to a rising transition.
- 1W1: W1 operation performed on a MTJ that initially contains “logic 1”. This sequence allows verifying the W1 operation since it applies both field-line current polarities.

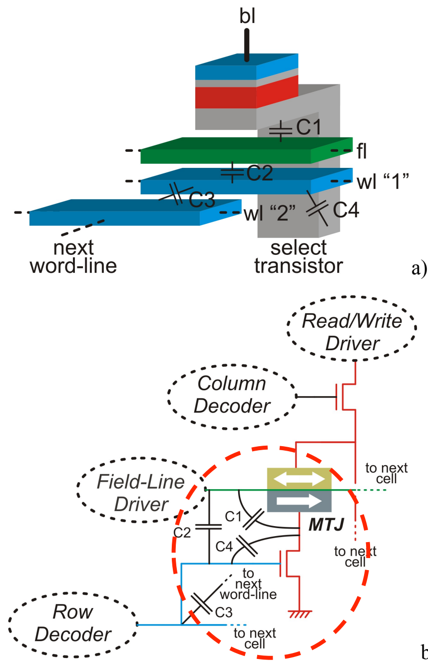


Figure 5. Capacitive defects injection
a) layout extraction and b) electrical modeling

For each capacitive defect, the capacitance range varies from the typical value to 10x the typical value. Larger capacitance sizes are unrealistic from process variations point of view. In those cases, other defect types, such as bridging defects, may appear.

A. MTJ bottom – Field-line (C1)

The current that passes through the MTJ is responsible for heating the device above its blocking temperature during write operations. This current is also important to retrieve the MTJ’s logic state during read operations.

Tables II and III summarize the simulated characteristics of $MTJ_{1,1,1}$ and $MTJ_{1,1,x}$ in the presence of a capacitive defect located between bottom metal of $MTJ_{1,1,1}$ and field-line labeled as C1. Gray lines represent typical C1 capacitance size. From these data, we observe that only MTJs sharing the same bit-line are barely affected by this defect when the defect size is up to 1fF. Consequently, there is no observed faulty behaviors for the considered defect range, i.e., from typical value to 10x the typical value.

TABLE II. $MTJ_{1,1,1}$ CHARACTERISTICS UNDER WRITE OPERATIONS

Operation	C1 (fF)	V (mV)	I (uA)	R (kΩ)	T (°C)	M
0W0	0.10	746.38	605.99	1.23	191.52	1
	1.00	746.36	605.91	1.23	191.58	1
1W0	0.10	744.68	606.92	1.22	194.09	1
	1.00	744.63	606.85	1.22	194.18	1
0W1	0.10	746.37 863.96	606.00 542.15	1.23 1.59	191.52 182.55	-1
	1.00	746.35 863.87	605.92 542.12	1.23 1.59	191.58 182.54	-1
1W1	0.10	744.64 862.99	606.93 542.66	1.23 1.59	194.11 183.93	-1
	1.00	744.66 862.99	606.85 542.54	1.23 1.59	194.18 183.99	-1

TABLE III. $MTJ_{1,1,x}$ CHARACTERISTICS UNDER WRITE OPERATIONS

Operation	C1 (fF)	V (mV)	I (uA)	R (kΩ)	T (°C)	M
0W0	0.10	746.45	606.00	1.23	191.45	1
	1.00	747.09	605.96	1.23	190.83	1
1W0	0.10	744.70	606.94	1.22	194.07	1
	1.00	745.24	606.96	1.22	193.68	1
0W1	0.10	746.44 864.01	606.00 542.10	1.23 1.59	191.45 182.33	-1
	1.00	747.12 864.43	605.95 540.93	1.23 1.59	190.79 179.95	-1
1W1	0.10	744.68 863.22	606.96 542.57	1.23 1.59	194.06 183.67	-1
	1.00	745.35 864.94	607.04 541.95	1.23 1.59	193.64 181.54	-1

In Figure 6, we show temperature profiles for W1 operations performed in one memory page under C1=1fF barely affecting the MTJs sharing the same bit-line ($MTJ_{1,0,1}$, $MTJ_{1,2,1}$ and $MTJ_{1,3,1}$). Note that MTJs are written row by row from $MTJ_{1,0,0}$ to $MTJ_{1,3,3}$. In addition, a non-catastrophic coupling effect is observed between $MTJ_{1,1,1}$ and $MTJ_{1,0,1}$, $MTJ_{1,2,1}$ and $MTJ_{1,3,1}$.

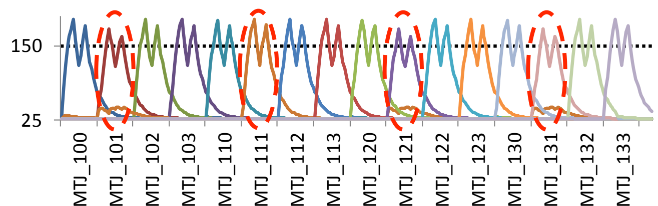


Figure 6. 16 bits W1 under capacitive defect (C1=1fF) temperature profile

B. Field-line – Word-line (C2)

Tables IV and V summarize the simulated characteristics of $MTJ_{x,1,0}$ and $MTJ_{x,2,0}$ under a capacitive defect located between field-line and word-line “1” labeled as C2. Simulations were performed using write sequences previously described.

In Figure 7, we show temperature profiles for W1 operation performed in one memory page under C2=10fF. This defect adds an extra delay when selecting/deselecting the affected word-line. If the defective word-line is half-selected, then operations performed on an MTJ on this word-line may be corrupted. Secondly, if the defective word-line remains partially selected when operations are applied elsewhere in the

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

TAS-MRAM array, then operations performed on a non-defective word-line may also be corrupted.

TABLE IV. $MTJ_{x,1,0}$ CHARACTERISTICS UNDER WRITE OPERATIONS

Operation	C2 (fF)	V (mV)	I (uA)	R (kΩ)	T (°C)	M
0W0	1.00	748.67	604.77	1.23	188.12	1
	10.00	769.01	572.79	1.23	128.30	1
1W0	1.00	746.68	605.83	1.22	191.28	1
	10.00	880.12	516.44	**1.70**	137.35	**-1**
0W1	1.00	748.67 864.50	604.77 541.85	1.23 1.59	188.12 181.52	-1
	10.00	769.01 767.92	572.79 593.69	1.23 1.59	128.30 163.63	-1
1W1	1.00	746.64 863.49	605.86 542.41	1.23 1.59	191.29 183.18	-1
	10.00	880.12 872.92	516.44 536.88	1.23 1.59	137.35 168.87	-1

TABLE V. $MTJ_{x,2,0}$ CHARACTERISTICS UNDER WRITE OPERATIONS

Operation	C2 (fF)	V (mV)	I (uA)	R (kΩ)	T (°C)	M
0W0	1.00	749.26	604.46	1.23	187.23	1
	10.00	773.40	589.91	1.23	147.24	1
1W0	1.00	747.62	605.36	1.22	189.79	1
	10.00	882.29	530.52	**1.70**	152,90	**-1**
0W1	1.00	748.90 864.58	604.63 541.82	1.23 1.59	187.77 181.42	-1
	10.00	769.79 867.45	591.87 541.63	1.23 1.59	153.26 171.08	-1
1W1	1.00	747.32 863.68	605.50 542.28	1.23 1.59	190.30 182.89	-1
	10.00	882.29 870.03	530.51 538.82	1.23 1.59	152.90 173.52	-1

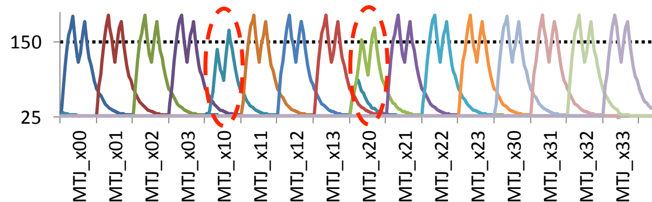


Figure 7. 16 bits W1 under capacitive defect (C2=10fF) temperature profile

The selection of word-line "1" is delayed in the presence of C2 as can be seen in the operation performed on $MTJ_{x,1,0}$. Next three operations performed on $MTJ_{x,1,1}$, $MTJ_{x,1,2}$ and $MTJ_{x,1,3}$ that share same word-line work properly as word-line "1" is fully selected. The operation performed on $MTJ_{x,2,0}$ has an undesired behavior as C2 delays the word-line "1" de-selection. Regardless defective word-line selection or de-selection only *1W0* sequence is not correctly performed. Such faulty behavior is modeled by a TF0 (Transition Fault 1 to 0).

C. Word-line – Word-line (C3)

Tables VI and VII summarize the simulated characteristics of $MTJ_{x,1,0}$ and $MTJ_{x,2,0}$, respectively, in presence of a capacitive defect located between word-line "1" and word-line "2" labeled as C3. Simulations were performed using write sequences previously described.

TABLE VI. $MTJ_{x,1,0/\,x,3,0}$ CHARACTERISTICS UNDER WRITE OPERATIONS

Operation	C3 (fF)	V (mV)	I (uA)	R (kΩ)	T (°C)	M
0W0	1.00	749.16	604.51	1.23	187.38	1
	10.00	755.41	576.71	1.23	147.72	1
1W0	1.00	747.10	605.60	1.22	190.58	1
	10.00	864.84	519.58	**1.70**	155,94	**-1**
0W1	1.00	749.16 864.77	604.50 541.71	1.23 1.59	187.37 181.30	-1
	10.00	755.42 853.37	576.70 546.17	1.23 1.59	147.71 169.08	-1
1W1	1.00	747.05 863.63	605.66 542.29	1.23 1.59	190,57 183.02	-1
	10.00	864.85 869.09	519.57 538.41	1.23 1.59	155.93 173.94	-1

TABLE VII. $MTJ_{x,2,0}$ CHARACTERISTICS UNDER WRITE OPERATIONS

Operation	C3 (fF)	V (mV)	I (uA)	R (kΩ)	T (°C)	M
0W0	1.00	758.25	599.58	1.23	173.55	1
	10.00	0	0	-	27.00	1
1W0	1.00	755.70	600.49	1.22	177.40	1
	10.00	0	0	-	27.00	**-1**
0W1	1.00	758.24 867.51	599.59 540.31	1.23 1.59	173.55 177.15	-1
	10.00	0 744.38	0 531.97	- **1.39**	27.00 92.59	**1**
1W1	1.00	755.75 866.08	600.51 540.96	1.23 1.59	177.41 179.40	-1
	10.00	0 862.29	0 477.05	- 1.80	27.00 100.78	-1

In Figure 8, we show temperature profiles for W1 operations performed in one memory page under C3=10fF. This defect adds an extra delay when selecting/deselecting both defective word-lines.

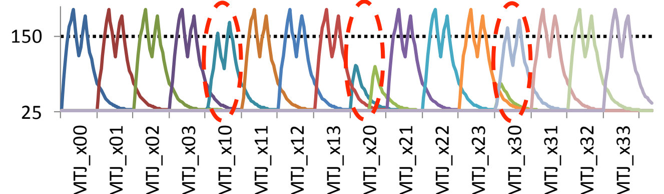


Figure 8. 16 bits W1 under capacitive defect (C3=10fF) temperature profile

As observed for C2, the selection of word-line "1" is delayed in the presence of C3 (operation performed on $MTJ_{x,1,0}$). Next three operations performed on $MTJ_{x,1,1}$, $MTJ_{x,1,2}$ and $MTJ_{x,1,3}$ that share same word-line work properly as their word-line is fully selected. The operation performed on $MTJ_{x,2,0}$ has an undesired behavior as C3 delays both word-line "1" de-selection and word-line "2" selection. Next three operations performed on $MTJ_{x,2,1}$, $MTJ_{x,2,2}$ and $MTJ_{x,2,3}$ work properly. The operation performed on $MTJ_{x,3,0}$ has an undesired behavior as C3 delays the word-line "2" de-selection. In addition to *1W0* operations, *0W1* operations are not correctly performed in presence of C3. Such faulty behavior is modeled by both TF0 and TF1 (Transition Fault 0 to 1).

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

D. Word-line – MTJ's bottom metal (C4)

Table VIII summarizes MTJ's simulated characteristics in presence of a capacitive defect located between word-line "1" and MTJ's bottom metal labeled as C4.

TABLE VIII. MTJS CHARACTERISTICS UNDER WRITE OPERATIONS

Operation	C4 (fF)	V (mV)	I (uA)	R (kΩ)	T (°C)	M
0W0	0.10	746.38	605.99	1.23	191.51	1
	1.00	746.34	605.92	1.23	191.60	1
1W0	0.10	744.69	606.90	1.22	194.09	1
	1.00	744.66	606.69	1.22	194.20	1
0W1	0.10	746.37 863.95	605.99 542.13	1.23 1.59	191.52 182.55	-1
	1.00	746.33 863.79	605.92 542.53	1.23 1.59	191.61 182.71	-1
1W1	0.10	744.67 862.94	606.92 542.68	1.23 1.59	194.10 183.95	-1
	1.00	744.63 862.97	606.71 542.66	1.23 1.59	194.20 184.00	-1

In Figure 9, we show temperature profiles for W1 operations performed in one memory page under C4=1fF.

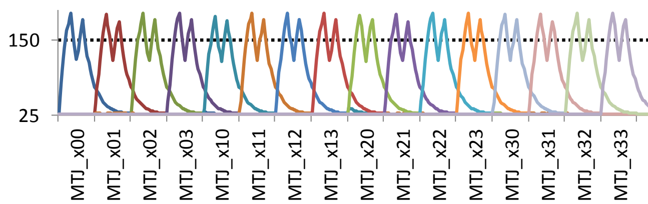


Figure 9. 16 bits W1 under capacitive defect (C4 =1fF) temperature profile

These electrical simulations show that C4 defect does not impact the functional operations of the TAS-MRAM.

V. CONCLUSION AND FUTURE WORK

In this paper, we have analyzed the impact of capacitive defect on the TAS-MRAM performance. Capacitive defect locations were extracted from the layout of the TAS-MRAM array and are simulated on a 16-words architecture enabling any sequences of read/write operations. Results have shown that writing operations may be affected by these coupling defects. Especially, we have demonstrated that some capacitive defects behave as transition faults while others, even if there is an effective coupling, do not change the functional operation of the TAS-MRAM. As future work, we plan to use these analyses results to guide the test phase by providing effective test algorithms targeting fault related to actual defects that may affect TAS-MRAM architectures.

ACKNOWLEDGMENT

This work has been funded by the French national research agency under the framework of the ANR-10-SEGI-007 EMYR (Enhancement of MRAM memory Yield and Reliability) project.

REFERENCES

[1] Semiconductor Industry Association, "International technology roadmap for semiconductors (ITRS)," 2011.

[2] C.L. Su, C.W. Tsai, C.W. Wu, C.C. Hung, Y.S. Chen, and M.J. Kao, "Testing MRAM for Write Disturbance Fault," in Proc. of IEEE International Test Conference, pp. 277-288, 2006.

[3] C.L. Su, R.F. Huang, and C.W. Wu, "MRAM Defect Analysis and Fault Modeling," in Proc. of IEEE International Test Conference, pp. 124-133, 2004.

[4] J. Azevedo, A. Virazel, A. Bosio, L. Dilillo, P. Girard, A. Todri, G. Prenat, J. Alvarez-Herault, and K. Mackay, "Impact of Resistive-Open Defects on the Heat Current of TAS-MRAM Architectures," in Proc. of Design Automation and Test in Europe, pp. 532-537, 2012.

[5] J. Azevedo, A. Virazel, A. Bosio, L. Dilillo, P. Girard, A. Todri, G. Prenat, J. Alvarez-Herault, and K. Mackay, "Impact of Resistive-Bridge Defects in TAS-MRAM Architectures," in Proc. of IEEE Asian Test Symposium, 2012.

[6] X. Aragones, J.L. Gonzalez, F. Moll, and A. Rubio, "Noise Generation and Coupling Mechanisms in Deep-Submicron ICs," IEEE Design & Test of Computers, vol. 19, no. 5, pp. 27-35, Sept.-Oct. 2002.

[7] D.D. Tang and Y.J. Lee, "Magnetic Memory – Fundamentals and Technology," Cambridge University Press, UK, 2010.

[8] M. El Baraji et al., "Dynamic Compact Model of Thermally Assisted Switching Magnetic Tunnel Junctions," Journal of Applied Physics, vol. 106, n° 12, 2009.

[9] W. Guo, "Compact Modeling of Magnetic Tunnel Junctions and Design of Hybrid CMOS/Magnetic Integrated Circuits," Ph.D. Thesis at Institut Polytechnique de Grenoble, 2010.

[10] Cadence Inc., Spectre, User Guide 2008.

Automatic Linking of Test Cases and Requirements

Thomas Noack
Berlin Institute of Technology
Daimler Center for Automotive IT Innovations (DCAITI)
Berlin, Germany
E-Mail: thomas.noack@dcaiti.com

***Abstract*—The paper proposes a 3-layered method which automatically creates trace links between test cases and reused requirements (test-links). While the first layer automates the manual test-link reuse, subsequent layers apply elaborate filter mechanisms. More specifically, Case-Based Reasoning is used in the third layer for detecting scenarios where test-link reuse is questionable. The proposed 3-layered method is explained with the help of a clarifying example.**

Keywords-Reuse; Requirements; Test cases; IBM DOORS

I. INTRODUCTION

Daimler uses the V-Model to manage methods and tools which guide the development process. Each vehicle series project passes the V-Model from the requirements stages over implementation to the test stages. The vertical integration defines which stages are performed by internal engineers and which stages are performed by external suppliers. Due to the low vertical integration in the automotive domain many engineers nowadays work mainly with engineering artifacts which are located in the upper stages of the V-Model. These artifacts are system requirements, test cases and trace links between them. The actual implementation is often done by suppliers.

Each new vehicle series inherits engineering artifacts from previously completed vehicle series projects. That means, reuse takes place from a source to a destination. The observation of the Daimler development process revealed several interesting facts. The reuse direction is horizontal from a source V-Model instance to a destination V-Model instance. Reusing system requirements is done by copying and adapting the requirements specification. Interestingly, test case reuse is not done via copying in practice. Instead, it is done by setting thousands of test-links from the existing test cases to the copied and adapted system requirements. This paper introduces a method to automate the complex task of setting the test-links between test cases and reused system requirements.

The paper is structured as follows: Firstly the Daimler specific DOORS®[1] modules and their interaction are introduced. After that the 3-layered method to reuse test-links is presented. The paper continues with a minimal example and related work. In the conclusion section, comments are made and future work is drawn.

II. DOORS® MODULES IN THE UPPER V-MODEL

Daimler uses DOORS® for requirements and test engineering. DOORS® manages specification documents in so called modules. Figure 1 shows the modules in the upper V-Model stages of the Daimler development process. The proposed method focusses on the relationship of the three following modules.

A. System Requirements Specification (SRS)

A vehicle is described by many SRS - one SRS for one system. Examples for systems are *Wiper Control* or *Outside Light Control*. The main engineering artifacts in SRS are vehicle functions. Examples for vehicle functions are *wash windshield* or *activate turn-signal right*. Each vehicle function is refined by specific (non-)functional requirements.

B. Test Specification (TS)

Test cases are the engineering artifacts in TS. A test case is characterized by test actions, pre- and pass conditions, assignments to test levels, test goals and many other properties. Test cases link to the corresponding requirements they verify. Each SRS has at least one corresponding TS.

C. Test Concept (TC)

The TC contains the test plan which defines, what must be tested when for which purpose. The test plan artifacts are: test object type (What?, e.g., *vehicle function*), test level (When?, e.g., *vehicle integration test*) and test goal (Which purpose?, e.g., *correct interaction on interfaces*). The TC defines which test-links must exist between TS and SRS in order to fulfill the test plan. Therefore, each test case in the TS is classified according to the test plan artifacts.

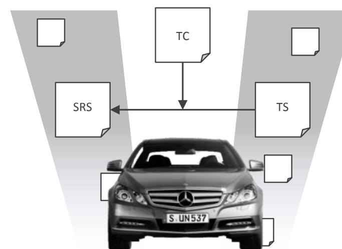


Figure 1. DOORS® modules in the upper V-Model

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

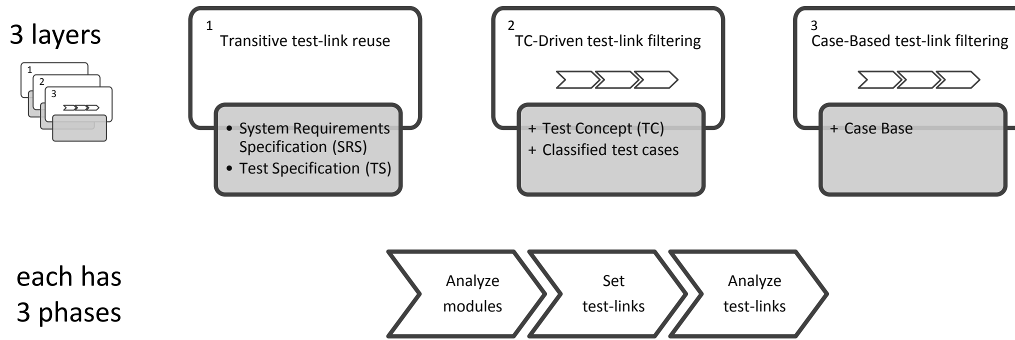


Figure 2. 3-layered method to reuse test-links

III. 3-LAYERED METHOD TO REUSE TEST-LINKS

This section describes the proposed 3-layered method for automating the reuse of test-links between TS and SRS. Figure 2 depicts the layers of the method.

Each layer consists of the same three phases. The specific tasks of each phase differ depending on the layers characteristics as indicated in Figure 3. A subsequenting layer enhances the phases of its predecessor with additional tasks. The general tasks performed in the three phases are as follows.

- *Analyze modules*: Extract information from the SRS_{Src} (Source), SRS_{Dst} (Destination), TS and TC.
- *Set test-links*: Set links from TS to SRS_{Dst} on the basis of the above analysis results.
- *Analyze test-links*: Assess the links and highlight the link status in SRS_{Dst} and in TS.

The first layer can be directly integrated into the Daimler development process because the process stipulates the existence of the involved modules SRS_{Src}, SRS_{Dst} and TS.

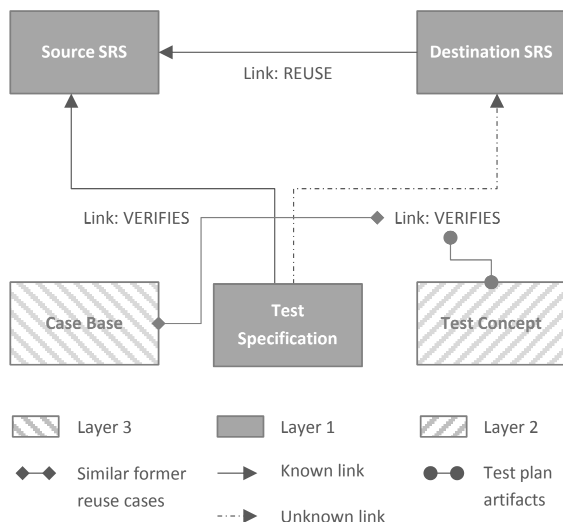


Figure 3. DOORS® modules needed by the layers

In Figure 3, the filled boxes and link arrows show the minimal reuse situation presumed by the first layer. When a new vehicle series project is launched, requirements are reused by copying the complete SRS_{Src} module. The resulting SRS_{Dst} is then adapted to the requirements of the new vehicle series. While the test-links from TS to SRS_{Src} do exist, the test-links from TS to SRS_{Dst} do not exist. The first method layer automatically sets the not existing test-links and highlights the link status in SRS_{Dst}.

The TC shown in the lower right corner of Figure 3 is the additional module needed by the second layer. The TC defines which test-links must exist in order to fulfill the test plan. The connection between TC and TS is established by classifying the test cases within TS according to the test plan artifacts of TC. By the virtue of taking test plan artifacts into account, the resulting test-links and highlighted requirements are much more comprehensive compared to the linking and highlighting of the first layer.

The Case Base shown in the lower left corner of Figure 3 is the additional module needed by the third layer. Case-Based Reasoning relies on two assumptions [2]: (1) similar problems have similar solutions and (2) similar problems occur continuously. Transfered to test reuse, these assumptions mean that (1) similar reuse situations result in similar reuse decisions and (2) similar reuse situations occur continuously. The cases of a Case Base are structural representations of previously applied knowledge [3]. Reuse knowledge is represented by differences between previous SRS_{Src}, SRS_{Dst}, TS and TC. The benefit of Case-Based filtering is that typical situations, which disable test-link reuse, can be recognized automatically.

Discussions with Daimler engineers led to an interesting conclusion. Case-Based Reasoning can, in the given context, only be used to detect situations, where test-link reuse is not possible. If, for example, the interface specification of a destination requirement changed, it can be assumed that a integration test case probably must be reviewed. On the other hand, it can not be automatically assumed that a test-link can be reused only because the interfaces did not change.

The layers and its phases are described as follows.

A. First layer: Transitive test-link reuse

1) Analyze modules: If a requirement is reused in SRS_{Dst}, it has a reuse-link to the corresponding requirement in SRS_{Src}. The textual similarity between each source and destination requirement is calculated and stored in SRS_{Dst}. New and heavily adapted requirements of SRS_{Dst} have no reuse-links to SRS_{Src}.

2) Set test-links: Test-links are reused transitively. That means, if a test case in TS verifies a requirement in SRS_{Src} and if this requirement has been reused by a requirement in SRS_{Dst}, then the test case in TS also verifies the requirement in SRS_{Dst}. If the destination and source requirement are textually identical, the test-link is reused. Otherwise, it must be reviewed by the test engineer in the next phase.

3) Analyze test-links: After the test-links have been set in the previous phase, SRS_{Dst} is analyzed. Three scenarios can occur for each destination requirement: (a) it is identical to the source requirement and hence a test-link can be reused directly (b) it has been changed slightly and, therefore, the test-link must be approved by the engineer (c) it has no test-link because either it has been changed heavily or it is a new requirement or the source requirement has no test-links.

B. Second layer: Test-Concept-Driven test-link filtering

1) Analyze modules: Based on the analysis of the TC, a 3-dimensional test plan cube is constructed. The cube dimensions are the test plan artifacts: test object type, test level and test goal. An example of a cube cell would be the triple (vehicle function, integration test, verify interface).

2) Set test-links: This phase enhances the first layer with a filtering mechanism enabled by the cube. In particular, the necessity of the test case is examined by passing the test case classification to the cube. Only if a test case is considered as needed by the cube, i.e., as needed to verify a test goal for a test object type on a test level, the test-link is set.

3) Analyze test-links: While the first layer can only make statements about the pure existence of test-links the second layer also considers test plan artifacts. Therefore, for each destination requirement the following more detailed scenarios arise: (a) a test case for a specific test object type is missing (b) a test case for a specific test goal is missing and (c) a test case for a specific test level is missing.

C. Third layer: Case-Based test-link filtering

1) Analyze modules: In this most sophisticated layer, Case-Based Reasoning (CBR) is utilized to filter test-links based on previous reuse experience. More specifically, a current reuse situation is constructed for each potentially reusable test-link by extracting relevant information from SRS_{Src}, SRS_{Dst}, TS and TC. The situations are then converted into structural case representations to enable similarity search in the next phase.

2) Set test-links: For each constructed current reuse situation, a similar case in the Case Base is searched. Therefore, similarity measures, as shown in [4], are applied. If a similar negative reuse case, i.e. where link reuse has been questionable, is found for a current test-link, its reuse possibility is also marked as questionable.

3) Analyze test-links: The third layer extends the previous layers with additional analysis possibilities with respect to not reusable test-links. For each classifying property more fine-grained scenarios arise, e.g., (a) interface has been changed thus an integration test case is questionable or (b) safety relevance has been changed thus a safety test case is questionable.

IV. EXAMPLE

Figure 4 depicts an example to show the application of all three layers. The following subsections describe, how the 3-layered method extends the current module landscape.

A. Current state of the modules

Currently, SRS_{Dst}, SRS_{Src} and TS are stipulated by the Daimler development process. While the TC has been rolled out lately, the Case Base is a new module which only exists conceptually. The SRS modules in Figure 4 contain the textual requirements Src_X and Dst_X and columns of their properties.The TS contains test cases which trace link to SRS modules.

B. Reuse relationship between SRS_{Src} and SRS_{Dst}

Src_{1+2} in SRS_{Src} and Dst_{1+2} in SRS_{Dst} are in a reuse relationship. Since Dst_1 has not been changed textually, it is 100% similar to Src_1. Dst_2 has been modified in order to adapt the changed needs of a new vehicle series. The textual similarity of Dst_2 and Src_2 is 80%. For further considerations we assume that 80% is within the borders of the reuse threshold. Dst_3 has changed heavily and the reuse relationship to Src_3 could not be detected technically. Dst_4 is a new requirement.

C. Transitive reuse of test-links

The test cases Test_X in TS have test-links to the requirements Src_X in SRS_{Src}. These test-links between Src_{1+2} and Test_{1+2} are reused to link to the corresponding requirements Dst_{1+2} in SRS_{Dst}. While the test-link between Test_1 and Dst_1 has been reused directly, the test-link between Test_2 and Dst_2 must be reviewed because the requirements Dst_2 and Src_2 are not identical. The test-link between Dst_3 and Test_3 is not set because Dst_3 and Src_3 have no reuse-link.

D. TC and classified test cases in TS

The TC stipulates that each test object of the type *vehicle function* must be verified on *integration* (Int) and *system* (Sys) test level. The test goal *functionality* must be verified by both, *integration* and *system* test. *Correct interaction on interfaces* has to be verified on the *integration* test level

Source SRS

Requirements	Interface	Reused?	Tested?
1 A function			Fully
Src 1		Yes	Yes
Src 2	Sys 1	Yes	Yes
Src 3		No	Yes

Destination SRS

Requirements	Interface	Reuses?	Similarity	Tested?	Test details
1 A function				Partly	- Not all requirements (1) - Test goal missing (2+3) - Interface - Test level missing (2+3) - Vehicle integration
Dst 1: Equal		Yes	100	Yes	
Dst 2: Slightly differ.	Sys 1 Sys 2	Yes	80	Review	- Req text changed (1) - Interface changed (3)
Dst 3: Very different		No		No	
Dst 4: New		No		No	

Test Concept (TC)

Test plan	Int	Sys
Test goals: Verify ..		
.. functionality	X	X
.. interface	X	
Test object type:		
Vehicle function	X	X

Case Base

Cases	Reuse?	Source	Destination
1 Cases			
Interface changed	Review	SRS.Interface = ["Sys 1"]	SRS.Interface = ["Sys 1", "Sys 2"]

Test Specification (TS)

Test cases	Test goal	Test level	Reuse?	Reuse documentation
1 Tests				
Test 1	Functionality	System	Yes	- Src 1 reused (1)
Test 2	Interface	Integr.	Review	- Src 2 reused but changed (1) - Interface changed (2+3)
Test 3	Configurability	Module	No	- Src 3 not reused (1)

Figure 4. Minimal example (DOORS®module state after running layer 3)

only. The test cases in the TS are classified by the test plan artifacts *test goal* and *test level*. $Test_3$ verifies the test goal *configurability*. Because this test goal is not considered by TC, a potential test-link from $Test_3$ would not be set.

E. Case Base

The requirements Src_2 and Dst_2 have a common property: *Interface*. While Src_2 has an interface to the system ['Sys 1'], the system dependencies of Dst_2 are ['Sys 1', 'Sys 2']. This current reuse situation is transformed to a case.

The search in the Case Base returns the case *Interface changed*, which is identical to the current reuse situation. The reuse decision *Review* of the case is applied to the current situation between $Test_2$ and Dst_2.

V. State of the Art

There are two possible sources from which reused test cases can originate: they come from already existent test cases or have been generated from reused test models. Because this work is located in the upper V-Model stages it clearly focusses on the first possible source.

Three works mainly inspired the 3-layered method. Geppert et al. describe, how textual test cases can be transformed to product line test cases [5]. Nebut et al. propose a requirement-based approach for testing product families [6]. They generate textual test cases from the so called Use Case Transition System, which is formed by trace links between Use Cases. Minor and Hanft use Case-Based Reasoning for reusing test cases by analyzing textual similarity [7].

VI. Conclusion and Future Work

This *Work in Progress* paper proposed the basic functionality of a 3-layered method which has been developed for supporting the industrial test case reuse process pragmatically. The first method layer has been piloted successfully in the automotive domain with real specification modules from the Wiper Control System and the Rain Closing System. Implementation details and field study results of each layer follow in future publications.

The proposed method does not only exclusively support the automotive domain. It is located in the upper V-Model stages and thus can be applied generally to each environment, where test cases and system requirements are connected by test-links.

References

[1] IBM, "Rational DOORS," http://www-03.ibm.com/software/products/us/en/ratidoor/ [Last access: 19/06/2013].

[2] D. B. Leake, "CBR in Context : The Present and Future," in *Case-Based Reasing: Expericences, Lessons and Future Directions*. MIT Press, 1996, ch. 1, pp. 1–35.

[3] R. Bergmann, J. Kolodner, and E. Plaza, "Representation in Case-Based Reasoning," *The Knowledge Engineering Review*, vol. 20, pp. 209–213, 2006.

[4] F. Brosius, "Distanz- und Ähnlichkeitsmaße (engl.: Distance and Similarity Measures)," in *SPSS 21*. mitp, 2013, ch. 31, pp. 693–709.

[5] B. Geppert, J. Li, F. Rössler, and D. M. Weiss, "Towards Generating Acceptance Tests for Product Lines," in *Proceedings of the 8th International Conference on Software Reuse*, Madrid, Spain, 2004, pp. 35–48.

[6] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "A Requirement-based Approach to Test Product Families," in *Proceedings of the 5th International Workshop on Software Product-Family Engineering*, Siena, Italy, 2003, pp. 198–210.

[7] M. Minor and A. Hanft, "The life cycle of test cases in a CBR system," *Advances in Case-Based Reasoning*, pp. 455–466, 2000.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

Using Filtering to Improve Value-Level Debugging of Verilog Designs

Bernhard Peischl
Softnet Austria
Graz, Austria
bernhard.peischl@soft-net.at

Franz Wotawa
Institute for Software Technology, TU Graz
Graz, Austria
franz.wotawa@ist.tuGraz.at

Naveed Riaz
Shaheed Zulfikar Ali Bhutto Institute
Islamabad, Pakistan
n.r.ansari@szabist-isb.edu.pk

Abstract— **In this article, we report on novel insights in model-based software debugging of hardware description languages (HDLs). Our debugging model allows one for exploiting failing and passing test cases by incorporating Ackermann constraints. This article reports on an empirical evaluation of the introduced models. The evaluation of our approach on the well-known ISCAS 89 benchmarks concerning single and dual-fault diagnoses clearly indicates that incorporating passing test cases into fault localization improves considerably the accuracy of the obtained diagnosis candidates.**

Keywords – hardware/software debugging, model-based debugging, source-level debugging, fault localisation

I. INTRODUCTION

This article reports on the most recent results in software debugging of Verilog designs. It is a major extension to previous research work that primarily reports on fault localization in Very High Speed Integrated Hardware Description Language (VHDL) [1]. Verilog [2], has a formal semantics and thus, it is amendable to research in verification and debugging, e.g., its synthesis semantics is formally specified in Gordon [3].

Most of the research in verification deals with the detection of faults and does not address the fact that debugging involves locating and correcting the fault. In detecting faults (software/hardware testing), we make use of numerous test cases for more than two decades. In the recent past, numerous test cases have been employed for localizing faults, e.g., in terms of employing spectrum-based diagnosis [4, 5, 6, 7, 8].

Spectrum-based techniques, however, allow one for logical reasoning at the level of dependencies and do not consider the semantics of the language in terms of value-level models. Consequently, there is a lack of research dealing with multiple test cases in conjunction with value-level models taking into account language semantics. This is noteworthy as we do have well-founded techniques that allow for considering whole test suites and – as shown in this article – there is solid empirical evidence that taking into account test suites improves the fault localization capabilities considerably.

Over the last 25 years, the Artificial Intelligence community has developed a framework for system diagnosis called model-based diagnosis (MBD). This framework is extremely general and covers a broad range of capabilities, including the isolation of faulty components and the handling of multiple fault locations [9, 10]. Harnessing these techniques in software engineering tools, may help considerably to master the development of complex circuits and software-enabled systems.

Since its well-founded theory, we rely on MBD, and employ the ISCAS 89 benchmark suite [11] to demonstrate the practical applicability of our novel models. Relying on an exhaustive evaluation, our insights clearly indicate that the incorporation of test suites (rather than only single test cases as for example in [12]) considerably contributes to locate accurately the root cause for detected misbehavior. According to our empirical evaluation using the ISCAS 89 benchmarks, with a couple of failing test cases (up to 5), we can exclude almost 94 percent of the statements and expressions of being faulty. By leveraging passing test cases, we can further rule out around half of the remaining 6% of the potentially erroneous code. In this article, we show how to incorporate passing test cases. In contrast to previous articles addressing this issue, we report on our most recent empirical evaluation on the ISCAS 89 benchmarks regarding the proposed filtering algorithm.

The next section gives a brief introduction to simulation, test and debugging of HDLs and afterwards (Section III), we discuss the debugging of sequential circuits. In Section IV, we show how to exploit passing test cases. Section V reports on practical experiences and the evaluation of the approach and Section VI concludes this article.

II. SIMULATION, TEST AND DEBUGGING

In designing circuits, a designer starts with an initial specification that primarily captures the functional requirements for the circuit being designed. Usually, this is followed by a detailed design on the register transfer level (RTL). Both designs are executable and thus are amendable to automated ver-

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

ification. In general, the RTL design is verified very thoroughly in terms of testing and various other analysis techniques, e.g., hazard analysis. Since there is a fixed window for start of production, these verification steps are typically conducted under time pressure and thus, the time for debugging – detecting, localizing, and repairing the misbehavior – is a critical process measure.

Typically, the design process iterates through several steps: Design and programming is followed by a simulation of the circuit. The outcome of the simulation is compared to the specification, that is, it is checked whether the waveform traces on a higher abstraction level (the specification) deviate from the waveforms obtained from the test run on the RTL level. Previous research work, carried out in the VHDL domain, gives an intuitive understanding on how to leverage MBD for fault localization in HDL designs1.

According to a study conducted at IBM Haifa, 50 to 80 percent of the overall development is attributed to verification activities, and localization and correction amounts to 35 percent of the design cycle [13]. Thus, particularly under local or temporal separation of the design and the test team, the automation of fault localization (and correction) is a sustainable topic for ongoing and future R&D work as it contributes to make the development process more efficient.

III. DEBUGGING SEQUENITAL VERILOG DESIGNS

The semantics of Verilog has been analyzed rigorously, and thus provides the necessary theoretical underpinning in language semantics and circuit synthesis. Gordon [3] provides a formal description of various semantic interpretations of Verilog like event-semantics and trace-semantics. In event-semantics (which is the semantics employed for fine-grained simulations), the change of a variable necessitates the recalculation of depending procedures.

In contrast to that, the trace semantics of Verilog computes solely the quiescent states at the end of a simulation cycle. For computing these quiescent values, each procedure is evaluated only once per cycle [3]. Procedures are evaluated in an order such that a procedure is not evaluated until all its driving procedures have been evaluated. In other words, the outputs of a procedure are computed only when all its inputs are known (or already computed). So, we build up our representation of the design by starting with processes solely dependent on known inputs and variables (e.g., the primary inputs, including clock). Afterwards, the outputs of these processes are attached to the list of already known inputs and variables. This process continues until all the procedures in the design are levelized [12]. In this way, we build up a chain of procedures and their inputs and outputs, thus allowing for an evaluation of all the variables used in the design at the end of the simulation cycle.

Synchronous sequential circuits change their states and output values at discrete instants of time, which are specified by the rising and falling edge of a clock signal. In other words, synchronous sequential circuits consist of multiple cycles. In electrical engineering, sequential circuits are often viewed as a sequence of connected combinational circuits. This can be done by selecting some connections and splitting them in two separated connections. One is the input and one the output. The output of a stage of a specific cycle is connected to the corresponding input of the next cycle.

We have adopted the same idea for providing an appropriate debugging model for sequential designs. Our representation can be broken into two phases, one in which latches change state, and one in which all the combinational blocks are evaluated. We effectively break the design at latches by treating the outputs of the latches as they were inputs and inputs of the latches as they were outputs.

In our representation, we first identify variables that we have to synthesize into latches. By splitting these variables and treating them as additional inputs and outputs, we ensure that our representation remains acyclic. Then, we levelize the graph according to the levelization strategy discussed above. Thus, we receive a sequence of procedures depicting the data flow from the given primary inputs to the primary outputs. Our next step is to unroll the sequential circuits to incorporate multiple cycles (input sequence length). We assume that we know the number of unrollings to be performed in advance. After the levelization of all the procedures, we create the component-connection model. This component-connection model [9, 10] represents our model at level 1 (cycle no. 1). For every component C, we attach a timestamp i during the creation of the model to ensure a unique identification. Thus Ci represents the instance of component C at cycle i. Thus, we make n copies of every component involved, where n is the total number of cycles or unrollings. So we create n instances for each component.

Diagnosis problem: A diagnosis problem considering circuit unrolling over n cycles is a triple *(SD, COMP,OBS)* where

$$SD = \bigcup_{i=1..n} SD_i \text{ where } SD_i \text{ is the system descry. for cycle i} \quad (1)$$

$$COMP = \bigcup_{i=1..n} C_i \text{ where } C_i \text{ are the components in cycle i} \quad (2)$$

and

$$OBS = \bigcup_{i=1..n} OBS_i \text{ and } OBS_i \text{ denote the obs. in cycle } i. \quad (3)$$

The above given definition captures a diagnosis model for a single test case (of length n). Given this definition, the diagnosis problem considering a test suite is given as follows:

Diagnosis problem, test suite: Given a test suite comprising the test cases $TC_1, TC_2, \ldots, TC_k$. Let the system description SD_j be the system description considering test case TC_j and let C_i^j be the instance of component C at cycle i in test case number j. Correspondingly, the sets OBS_i^j denote the observations in cycle i of test case TC_j. The diagnosis problem $(SD^*, COMP^*, OBS^*)$ considering this test suite is given as follows:

$$SD^* = \bigcup_{j=1..k} SD_j \cup \{\neg AB(C_0^{\,j}) \to \neg AB(C_1^j) \wedge \neg AB(C_2^j) \wedge ... \quad .... \wedge \neg AB(C_n^j)\} \quad (4)$$

$$COMP^* = \bigcup_{j=1..k, i=0..n} C_i^j \quad (5)$$

$$OBS^* = \bigcup_{j=1..k, i=1..n} OBS_i^j \quad (6)$$

As passing testcases do not cause a logical contradiction, we do not obtain conflicts from passing testcases considering the diagnosis model for a test suite (SD*, COMP*, OBS*).

IV. EXPLOITING PASSING TESTCASES

To illustrate the potential of using passing test cases to locate the root cause for detected misbehavior we continue with a simple example.

assumption	in1	in2	out	inter	verdict
AB(not), ¬AB(xor)	1	0	1	0	fail
AB(not), ¬AB(xor)	0	0	1	1	pass

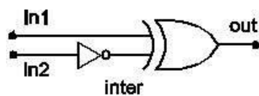


Figure 1: Passing and failing testcases and part of a circuit.

Figure 1 illustrates a part of a circuit an exclusive or and a NOT gate together with a passing and failing test case. We further assume that the circuit is faulty, that is, our test suite has identified misbehavior and we obtain both components (the exclusive OR and the NOT gate) as possible diagnosis candidates.

Suppose we have the test cases given in Figure 1. Considering the first (failing) test case in the first line, and assuming the NOT gate to be abnormal but the exclusive OR gate to be correct, we can deduce that signal inter becomes 0. However, under the same assumption, the passing test case in line 2, forces the value of inter to become 1. We immediately see that the NOT gate is required to map the signal inter to 0 and to 1 for the same input value in2=0. Obviously, no deterministic component can fulfill this requirement. Thus, the NOT gate can no longer be considered as a valid diagnosis candidate. To our best knowledge, the authors of [14] were the first who used this idea for discriminating diagnosis candidates. Unfortunately, the article gives no further insights whether the technique can be employed in practice as the authors do not provide an empirical evaluation to evaluate scalability and the improvement with respect to accuracy.

In the following, we propose an extension to that which, under absence of structural faults, allows one for taking advantage of passing test cases. As passing test cases does not yield to additional conflicts, we capture the specific information about diagnoses in terms of Ackermann constraints [22, 23]. By adding these consistency constraints we incorporate the fact that the same combination of input values applied to a deterministic component C produces the same output for every instance of C. This allows for exploiting the many test cases that typically do not reveal a fault. The system description with Ackermann constraints SDA is given as follows:

System description with Ackermann constraints: Let TC_p be a set of passing test cases form a test suite *TC*, let $in(C_i) = \{ i_{Ci}^1, ..., i_{Ci}^m \}$ denote the inputs of component C_i, let $out(C_i) = \{ o_{Ci}^1, ..., o_{Ci}^n \}$ denote the outputs and let SD^* denote the system description of a diagnosis problem considering a test suite. The system description with Ackermann constraints SD_A is given by,

where, i≠j and i,j denote indices of the passing test cases. As we will show in the next section, Ackermann constraints increase the complexity of the model considerably.

$$SD_A = SD^* \cup CON_A, \quad (7)$$

$$CON_A = \neg AB(C_i) \wedge \forall_{l=1}^m i_{ci}^l = i_{cj}^l \to \forall_{p=1}^n o_{ci}^p = o_{cj}^p \quad (8)$$

Therefore, we used a post processing technique proposed by the authors of [21]. As shown at the end of this section, filtering allows one for iteratively applying the Ackermann constraints to the obtained diagnoses. Instead of compiling the constraints into the debugging model, we apply the constraints in terms of a dedicated post-processing phase.

Filtering refers to discarding certain diagnoses by taking advantage of further test cases TC_i. A diagnosis Δ states that $\Delta \cup SD \cup TC_i \cup \{\neg AB(C) \mid C \in COMP \setminus \Delta\}$ is consistent. This implies that there is a replacement, that is, there exists a function *replace(C)* for every component $C \in \Delta$ that allows for repairing the program for the given test case. The function *replace(C)* allows for producing the correct output values for the considered test case. However, considering a test suite such a replacement does not exist for all test cases in the test suite TC necessarily.

Since all components *COMP \ Δ* are assumed to behave correctly, we can compute the input values *in(C)* and *out(C)* for every component *C* from *Δ* (employing forward propagation). According to this computed input/output relation, the component *C* may be required to map the same input- to different output values. This corresponds to an inconsistency and the specific diagnoses *AB(C)* is not repairable wrt. the specific test case. As there is no function *replace(C)* as stated previously, the component *C* can be removed from the set of diagnosis candidates. In this vein, we evaluate the Ackermann constraints in an iterative way by checking for different input values for a certain output value.

Algorithm 1 (Filtering): Let Δ denote a set of diagnosis candidates and let TS be a test suite.
1. *For all D ∈ Δ do*
2. *For all test cases TC_i ∈ TC do*
   a. *Let i_{Di} denote the input values and let o_{Dj} denote the output values of component D by assuming* $AB(D) \wedge \{\neg AB(C) \mid C \in COMP \setminus D\}$
   b. *If there exits i,j, i≠j, such that* $i_{Di} = i_{Dj} \wedge o_{Di} \neq o_{Dj}$ *then remove D from Δ*
3. *return Δ*

Figure 2: Exploiting passing testcases via filtering.

Claim: Algorithm 1 applies the Ackermann constraints CON_A to a set of single-diagnosis candidates.
After applying Algorithm 1 to the set of single-fault diagnosis candidates, there is no component D at which we obtain different input values for a certain output value. Thus, we conclude that

$$\neg\exists i, j, i \neq j \bullet (\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \wedge (\forall_{p=1}^{n} o_{Di}^{p} \neq o_{Dj}^{p}) \quad (9)$$

$$\forall i, j, i \neq j \bullet \neg(\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \vee (\forall_{p=1}^{n} o_{Di}^{p} = o_{Dj}^{p}) \quad (10)$$

$$\forall i, j, i \neq j \bullet (\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \rightarrow (\forall_{p=1}^{n} o_{Di}^{p} = o_{Dj}^{p}) \quad (11)$$

Algorithm 1 (Figure 2) thus imposes the Ackermann constraints on the set of single-fault diagnosis candidates. Therefore, for our approach evaluation we therefore took advantage of the filtering algorithm presented previously.

V. PRACTICAL EXPERIENCES AND EVALUATION

With a series of our most recent experiments we pursue the goal to evaluate the discriminating capabilities of several test cases on sequential circuits, the response time (and thus the computational complexity on a technical level) and the effect of the filtering technique.

We conducted our experiments on a Dell Power Edge 1950 II - 2x Quad Core with 2.0 GHz and 10GB of RAM. For computing diagnoses, we relied on the extension of Reiter's algorithm described in [15]. Note that, for the efficient computation of diagnoses, we convert the rules capturing the language semantics (discussed in [16]) into a specific Horn-like encoding [17]. As the computation of conflict sets is a time critical issue, the (minimal) conflict sets are computed according to the procedure explained in [17].The diagnosis engine and the proposed extension are implemented in the Java programming language.

Our debugging tool parses the Verilog code, builds up the model as described in this article and converts a test suite to the logical representation [16]. Afterwards, the tool computes diagnosis candidates in increasing order of cardinality and visualizes the results by highlighting the corresponding statements, expressions or operators.

A. Time Complexity of Computing Diagnosis

For our empirical evaluation, we use a Horn-like encoding of the rules presented herein. By relying on this encoding we make use of an efficient procedure to compute all minimal conflicts [17]. From the obtained conflicts, we retrieve diagnoses by computing the minimal hitting sets in increasing order, where for practical purposes, primarily single- and double-fault diagnoses are of interest. In general, searching for all diagnoses has a worst time complexity of the order O(|MODES|*|COMP|s), where |MODES| is the number of fault modes, |COMP| is the number of components and s is the maximal size of the diagnoses [18]. Since we use two fault modes (AB(C) and ¬AB(C)) and search for single and double fault diagnoses, our worst time complexity is of the order O(|COMP|2). Note that we consider the components in every cycle as independent and thus the number of components increases with the length of the test case. However, the average running time complexity is much better because diagnoses with smaller size (particularly single-fault diagnoses) are more likely than diagnoses with bigger size. For example, finding all single diagnoses is of order O(|COMP|) assuming the decision procedure can be executed in unit time.

B. Test Suite Generation

We obtained the test suite by injecting a single-fault (respectively a dual-fault for the second series of experiments) into the RTL design. Afterwards, we identified the faults in terms of running a simulation until we obtained five test cases revealing the introduced fault. In some (rare) cases, for example for the circuit s444, we were not able to find five test cases and stopped this process earlier (see Figure 3). The faults are introduced in a random way by picking a statement from every circuit and replacing this statement by another statement. That is, for every circuit, we replaced an arbitrary statement with a structurally equivalent statement (same no. of input parameters). For example, in a specific circuit we randomly selected a NOR statement and replaced it by an AND statement. Furthermore, we implicitly removed/added negations as we substituted a logical statement by the negated counterpart (e.g., NAND by AND vice versa). These error types are not necessarily complete wrt. functional errors, but as they are believed to be common in the design process, we capture the most com-

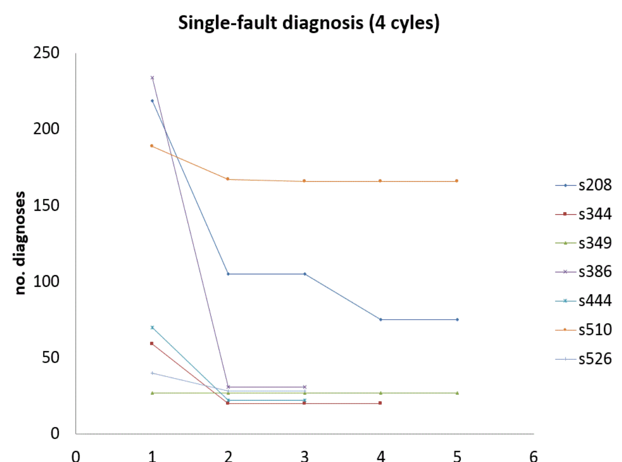


Figure 3: No. of obtained single-fault diagnoses for the ISCAS 89 benchmark (4 cycles).

mon scenarios [20]: (1) Mistakenly replacing one gate by another gate with the same number of inputs and (2) incorrectly adding or removing a gate.

All empirical evaluations are conducted on the Verilog RTL version of the ISCAS 89 benchmark suite [11]. Further, the gate-level representations of the ISCAS 89 benchmarks have been used to obtain the correct waveform traces since our simulator allowed only for simulation of gate-level circuits. A detailed analysis including the results for the specific circuits can be found in [16]. In the following, we summarize the major results. In this article, we summarize the work presented in [16] and present novel results regarding the incorporation of passing tests alongside with first empirical results.

C. Empirical Evaluation and Discussion

In our experimental setting, we assumed that an engineer only knows the correct values of the primary inputs for every simulation cycle and the outputs at the end of the final simulation cycle. That is, the traced variables correspond to the primary inputs vin for every instant of time (vin, valin, t), t=1..n, together with the primary outputs (vout, valout, n) at time n and thus, the observations are given in terms of the primary input variables for every cycle and the primary output variables at the end of the simulation cycle (i.e., at time point n, where n is the length of the test case). To evaluate the impact of the temporal unfolding of the circuit, we conducted experiments with four and eight simulation cycles relying on the well-known ISCAS 89 benchmark suite.

First, the figures underpin the findings discussed in previous research papers [19]. The number of single diagnoses being obtained depends from both, the concrete test case being applied and the structural complexity of the program being considered. Second, as Figures 3 and 4 illustrate – even with only a couple of test cases (in our case up to 5) – the number of obtained diagnoses can be reduced significantly when compared to the experiment with solely a single test case. Remarkably, the random fault introduced in circuit s510 yields to a significant number of diagnoses and thus higher response times when compared to the remaining circuits. It appears that (1) the structural complexity, (2) the specific error being introduced and the (3) specific test cases identifying the introduced faults result in a (at least in relation to the other circuits) computationally expensive problem. On average, we obtained 74(123) single-fault diagnoses and 44(70) faulty lines in the source code when unfolding the circuit for 4(8) instances of time. Remarkably, a designer can exclude over 90 percent of the source code from being faulty (93,6 percent for 4 cycles and 92,5 percent for 8 cycles of unfolding).

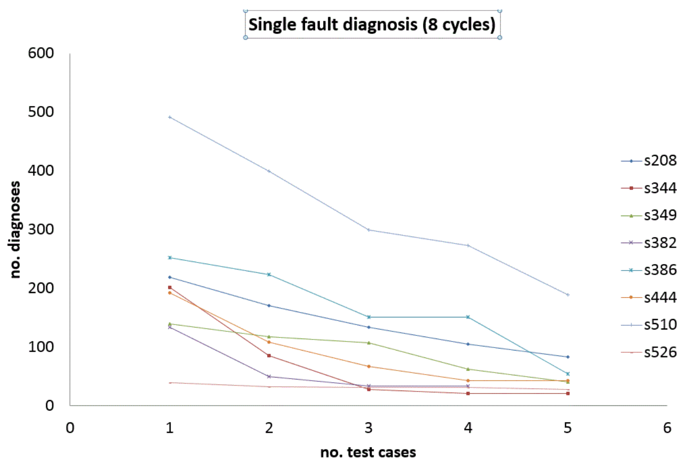


Figure 4: No. of obtained single-fault diagnoses (ISCAS 89, 8 cycles).

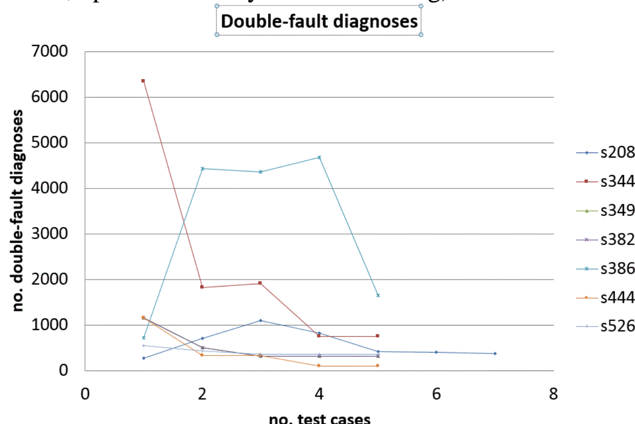


Figure 5: No. of obtained dual-fault diagnosis (ISCAS 89, 4 cycles).

Figure 5 outlines further empirical results. We obtained these results from the ISCAS 89 benchmark suite considering dual-fault diagnoses as well. When considering dual-fault diagnoses, the no. of diagnosis candidates does not necessarily decrease monotonically with the increasing set of test cases.

However, our experiments revealed that for most of the circuits, the obtained number of fault candidates decreases monotonically with an increase in the size of the test suite. Together with the results for single-fault diagnoses, this gives empirical evidence that the additional cost in the running time, pays off in terms of a higher accuracy in the obtained diagnosis candidates. In [15], we present novel algorithms and an analysis on scalability and the corresponding running times.

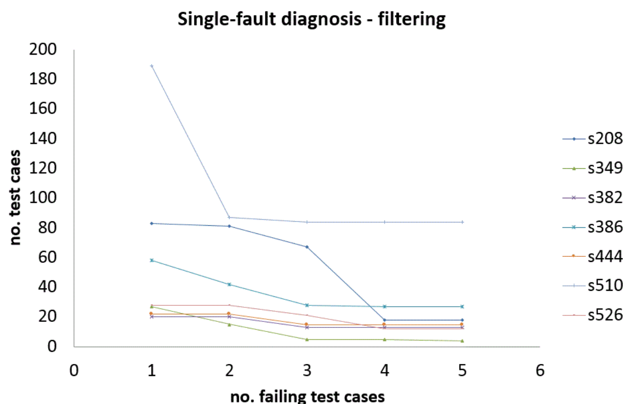


Figure 6: No. single-fault diagnoses when using the filtering algorithm (4 cycles).

Figure 6 summarizes the results on a further series of experiments incorporating the filtering algorithm. To our best knowledge, the filtering approach has never been subject to an empirical evaluation. When compared to Figure 3, one can see that exploitation of passing test cases contributes to accurately isolate the real cause of misbehavior.

VI. Conclusion

In this article, we briefly discuss the simulation-driven design process with hardware description languages (HDLs) and point out the importance of fault localization techniques. Afterwards, we introduce a model extension that allows one for exploiting failing and passing testcases. Failing testcases results in conflicts, and thus it contributes to locate the fault in an accurate manner. To exploit the numerous passing test cases, we introduce Ackermann constraints and establish a relationship to the filtering technique proposed earlier. Our empirical evaluation on the ISCAS 89 benchmark suite demonstrates that the proposed technique is practically feasible and considerably contributes to locate the real cause of misbehavior. According to our experiments using the ISCAS 89 benchmarks, on average, we can exclude almost 94 per cent of the statements and expressions from being faulty making use of up to 5 failing test cases per circuit. By leveraging passing test cases, we are able to rule out around half of the remaining 6 per cent of the potentially erroneous code. These results motivate research on value-level models for debugging HDL designs. Future research should apply the proposed techniques to even bigger circuits (e.g., using more recent benchmarks, etc.) and investigate the relationship between filtering and Ackermann constraints under presence of multiple-fault diagnoses.

References

[1] Z. Navabi, VHDL Analysis and Modeling of Digital Systems, McGraw-Hill, New York, 1993.

[2] IEEE Standard Verilog Language Reference Manual LRM Std 11364-1995, Institute of Electrical and Electronics Engineers, Inc. IEEE, 1995.

[3] M. J. C. Gordon, “Relating event and trace semantics of hardware description languages”, The Computer Journal, 45(1), 2002, pp. 27–36.

[4] R. Abreu, P. Zoetewei, van A. J. C. van Gemund, “On the Accuracy of Spectrum-based Fault Localization”, Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007, TAICPART-MUTATION 2007, vol., no., 10-14 Sept. 2007, pp. 89-98.

[5] B. Baudry, F. Fleurey, Y. Le Traon, “Improving test suites for efficient fault localization”, In Proceedings of the 28th international conference on Software engineering (ICSE '06), ACM, New York, NY, USA, 2006, , pp. 82-91.

[6] D. Hao, L. Zhang, T. Xie, H. Mei, and Jia-Su Sun, 2009, “Interactive fault localization using test information”, J. Comput. Sci. Technol. 24, 5, September 2009, pp. 962-974.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. I. Jordan, “Scalable statistical bug isolation”, In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05), ACM, New York, NY, USA, 2005, pp. 15-26.

[8] Y. Yu, James, A. Jones, M. J. Harrold, “An empirical study of the effects of test-suite reduction on fault localization”, In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 2008, pp. 201-210.

[9] R Reiter, A theory of diagnosis from first principles, Artif. Intell. 32, April 1987, pp. 57-95.

[10] J. de Kleer, A. K. Mackworth, R. Reiter, “Characterizing diagnoses”, In Proceedings of the National Conference on Artificial, Intelligence (AAAI), Boston, Aug. 1990, pp. 324–330.

[11] F. Brglez, D. Bryan, K. Kozminski, “Combinational Profiles of Sequential Benchmark Circuits”, IEEE International Symposium on Circuits and Systems, 1989, pp. 1926-1934.

[12] B. Peischl and F. Wotawa, “Automated Source-Level Error Localization in Hardware Designs”, IEEE Design and Test of Computers, January/February, 2006, pp. 8-19.

[13] G. Auerbach, M. Moulinn, B. Jobstmann, R. Bloem, A. Cimatti, M. Roveri, PROSYD: Property-Based System Design, Deliverable 2.1/1, May 2005, PROSYD Technical Report, FP6-IST-507219.

[14] O. Raiman, J. de Kleer, V. Saraswat, and M. Shirley, “Characterizing non-intermittent faults”, In Proceedings AAAI, Anaheim, Morgan Kaufmann, July 1991, pp. 849–854.

[15] B. Peischl, N. Riaz, F. Wotawa, “Advancements in Automated Debugging of Verilog Designs”, Submission to the Applied Artificial Intelligence Journal in preparation.

[16] B. Peischl, N. Riaz, F. Wotawa, “Automated Debugging of Verilog Designs”, International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Sept. 2012, Vol. 22, No. 5, World Scientific, pp. 695-724.

[17] B. Peischl and F. Wotawa, “Computing Diagnosis Efficiently: A Fast Theorem Prover for Propositional Horn Theories”, In Proceedings of the 14th International Workshop on Principles of Diagnosis (DX-03), Washington DC, June 2003, pp. 175-180.

[18] F. Wotawa, Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages, PhD thesis, Technische Universität Wien, 1996.

[19] G. Friedrich, M. Stumptner, F. Wotawa, “Model-based diagnosis of hardware designs”, Artif. Intell. 111(1-2), 1999, pp. 3-39.

[20] D. Nayak, D. M. H. Walker; “Simulation-based design error diagnosis and correction in combinational digital circuits”, VLSI Test Symposium, Proceedings of the 17th IEEE Test Symposium (VIS 99), 1999, pp. 70-79.

[21] F. Wotawa, “Debugging hardware designs using a value-based Model”, Applied Intelligence, 16(1), 2002, pp. 71–92.

[22] W. Ackermann, Solvable Cases of Decision Problems, North Holland, 1954.

[23] S. Staber, G. Fey, R. Bloem, and R. Drechsler, “Automatic fault localization for property checking”, In E. Bin, A. Ziv., and S. Ur, editors, Second International Haifa Verification Conference (HVC 2006), Haifa, Israel, October 2006, Springer-Verlag, LNCS 4383, pp. 50-64.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

Towards an Integrated Methodology for the Development and Testing of Complex Systems

Philipp Helle, Wladimir Schamai
EADS Innovation Works
Hamburg, Germany
Email: {philipp.helle,wladimir.schamai}@eads.net

***Abstract*—This paper reports on a framework for the development and testing of complex systems. The framework provides a meta-model for the description of systems at different levels of abstraction which is used as a basis for the combination of model-based testing (MBT) techniques for automated test case generation with executable requirement monitors that continuously observe the status of the System under Test (SuT) during test execution. The overall goal is to reduce the total development and testing effort for complex systems. This is accomplished by enabling a high degree of automation and reuse of engineering artefacts throughout the systems engineering lifecycle.**

***Keywords*—Model-based Systems Engineering, Model-based Testing, Monitor-based Testing, SysML.**

I. Introduction

The ever-increasing complexity of products has a strong impact on time to market, cost and quality. Products are becoming increasingly complex due to rapid technological innovations, especially with the increase in electronics and software even inside traditionally mechanical products. This is especially true for complex, high value-added systems in the aerospace and automotive domain - the methodology was developed and is therefore embedded in an aeronautic context but generally is independent of a specific domain - that are characterized by a heterogeneous combination of mechanical and electronic components. System development and integration with sufficient maturity at entry into service is a competitive challenge in the aerospace sector. Major achievements can be realized through efficient system testing methods.

"Testing aims at showing that the intended and actual behaviours of a system differ, or at gaining confidence that they do not. The goal of testing is failure detection: observable differences between the behaviours of implementation and what is expected on the basis of the specification"[1].

The typical testing process is a human-intensive activity and as such it is usually unproductive and often inadequately done. It requires human test engineers to manually write test cases. A test case contains a series of test inputs and expected results. Nowadays, the test execution especially on lower levels of testing is largely automated. Nevertheless, this process is cumbersome and costly. Therefore, testing is one of the weakest points of current development practices. According to the study in [2] 50% of embedded systems development projects are months behind schedule and only 44% of designs meet 20% of functionality and performance expectations. This happens despite the fact that approximately 50% of total development effort is spent on testing [2], [3]. This shows the importance and desirability of reducing test effort by advances in the testing methodologies.

Testing needs to be applied as early as possible in the lifecycle to keep the relative cost of repair for fixing a discovered problem to a minimum. This means that testing should be integrated into the lifecycle model so that each phase in the development contributes to the verification of the product as Figure 1 shows. Laycock claims that "the effort needed to produce test cases during each phase will be less than the effort needed to produce one huge set of test cases of equal effectiveness on a separate lifecycle phase just for testing"[4].

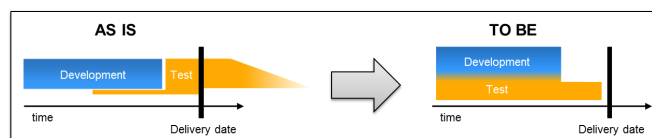


Fig. 1: Envisaged process change

This paper reports on a framework to further automate the system testing process. It is a continuation of the work earlier reported in [5]. The framework provides a meta-model for the description of systems on different layers of abstraction and combines model-based testing (MBT) techniques for automated test case generation based on a whitebox SysML model of the system with executable requirement monitors that continuously observe the status of the System under Test (SuT) during test execution. The overall goal is to achieve a high degree of automation and reuse of engineering artefacts throughout the systems engineering lifecycle.

Paper structure: First we present background information on SysML, MBT and monitor-based testing (Section II) before we will explain the methodology in detail (Section III). Finally, we propose a number of ideas for future research (Section IV) and close with a summary of the current status (Section V).

II. Background

This section provides background information on SysML, Model-based testing, Monitor-based testing and related work.

A. SysML

The Unified Modeling Language (UML) [6] is a standardized general-purpose modelling language in the field of software engineering and the Systems Modeling Language (SysML) [7] is an adaptation of the UML aimed at systems engineering applications. Both are open standards, managed

and created by the Object Management Group (OMG), a consortium focused on modelling and model-based standards.

SysML is not a methodology, i.e., it does not define what steps need to be performed in what order and which diagrams should be used for which step. Estefan [8] provides an overview of existing methodologies used in industry, some of which use UML-based languages. SysML is a graphical modelling language, i.e., diagrams are used to create and view model data. However, the graphical representation is decoupled from the actual model data. The model data and its graphical representation are typically stored in different files in UML/SysML tools.

Neither UML nor SysML define complete model execution semantics in their core specification. This is different from modelling and simulation languages, such as Modelica [9], which specify the syntax (textual notation) as well as the execution semantics. However, work is underway to resolve that [10], [11], [12]. In the mean time, SysML tool suppliers often provide their own execution semantics [13], so it is possible to include action code into models, generate code from the models and then execute them.

B. Model-based testing

The term MBT is widely used today with slightly different meanings. Surveys on different MBT approaches can be found in [1], [14], [15]. The most simple one is that ”Model-based testing (MBT) relates to a process of test generation from an SuT model by application of a number of sophisticated methods”[16].

Model-based testing is a variant of testing that relies on explicit behaviour models that encode the intended behaviour of a system and possibly the behaviour of its environment. The use of explicit models is motivated by the observation that traditionally, the process of deriving tests tends to be unstructured, barely motivated in the details, not reproducible, not documented, and bound to the creativity and expertise of single engineers. The idea is that the existence of an artefact that explicitly encodes the intended behaviour can help mitigate the implications of these problems [1].

Intensive research on MBT and analysis has been conducted in recent years, and the feasibility of the approach has been successfully demonstrated, e.g., in [17], [16]. Yet, Boberg [18] shows that most studies apply model-based testing on the component level, or to a limited part of the system while only few studies focus on the application of the technique on the system or even aircraft level. The main difference being that the goal of a system integrator such as Airbus is not to produce code but to provide a high quality specification that can be handed over for implementation to a supplier. Giese [19] explains that this slow adoption is not only due to scalability reasons but he also claims that ”to benefit from formal verification and early simulation, a model must be precise and detailed with respect to all aspects that are the subject of verification. This can usually be carried out in the detailed design phase at the earliest”[19].

A major distinction between the different available MBT approaches can be made by looking at the source of the generated test cases [19]. Some approaches rely on separate explicit test models that are disjunct from the system or specification model, as depicted by Figure 2 while other approaches don’t make that distinction and generate test cases from the defined system behaviour as shown by Figure 3.

The usage of explicit test models reflects the different objective (validation vs. solution) and point of view (tester vs. implementer) in creating a test model rather than a specification model [20]. A test model is a model representing all possible stimulations of input of the system interacting in various usage contexts and normally also includes verification points stating what is a correct response from the system to an input and what not. It thereby follows a tester’s view who also has to think of how to combine the possible input stimuli of a system to achieve a high confidence in its correctness.

The main benefit of this approach is the degree of independence it naturally entails between the generated test cases and the system. The generated test cases can thus be used directly to test any form of the SuT, either a model or the implementation. Additionally, as the test model is not a part of the design it can be optimised for validation and verification purposes thereby increasing the chance to uncover defects that are outside the focus of the design artefacts [19]. A drawback of the approach is that there are two models that have to be kept consistent with the requirements at all time which requires further effort.

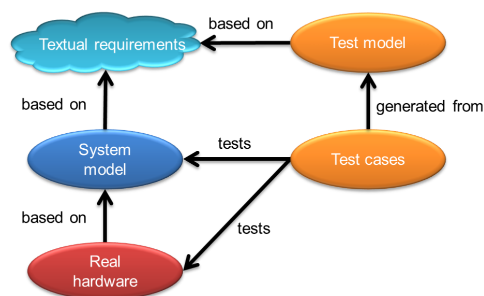


Fig. 2: Model-based testing using explicit test models

One example for an approach that does not rely on explicit models is the work from Lettrari [21] that is the basis for the commercially available IBM Rational Rhapsody Automatic Test Generator (ATG) tool. Test cases are generated from a behaviour model of the SuT using model coverage as test selection criteria. Automated test case generation uses constraint based symbolic execution of the model and search algorithms.

The main benefit is that the approach does not require the creation and maintenance of a separate test model. On the other hand, since the test case generation is not guided by a test engineer it cannot distinguish between ”good” and ”bad” test cases. The only goal for the generator is to achieve a high degree of model and/or code coverage by generating stimuli that force the executable system model to visit all states and transitions and call all functions of the system’s components. Furthermore, there is no independence between the generated test cases and the system model. This means that the test cases cannot be used to test the model they were generated from if the test success criteria is that the observed behaviour and the test case behaviour are the same.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

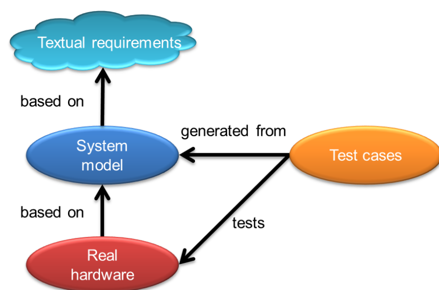


Fig. 3: Model-based testing using design/specification models

C. Monitor-based testing

The idea for formalizing a natural language requirement statement into a requirement monitor is similar to the monitor concept used in runtime verification [22], [23]. Like in runtime verification, the main purpose of the requirement violation monitor is to detect requirement violations without intervening into the analyzed system. A more formal definition states, that ”Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property”[22].

Runtime verification itself deals with the detection of violations of correctness properties. Thus, whenever a violation is observed, it typically does not influence or change the programs execution, say for trying to repair the observed violation. Checking whether an execution meets a correctness property is typically performed using a monitor. In its simplest form, a monitor decides whether the current execution satises a given correctness property by outputting either yes/true or no/false [22].

D. Related Work

In [24], Artho et. al. propose a method for combining test case generation and runtime verification for software systems. In their framework they combine automated test case generation, which is based on a systematic exploration of the input domain of the tested software system using a model checker that is extended with symbolic execution capabilities with runtime verification techniques, that monitor execution traces and verify them against properties expressed in a temporal logic notation. They include further capabilities for the analysis of concurrency errors, such as deadlocks and data races. The paper also provides a description of the application of the method using a NASA rover controller.

While similar on an abstract level, our work differs from the work by Artho et. al. in some major points. Firstly, the test oracles are written as temporal logic formulas whereas we use SysML for both the modelling of the system as well as the requirement monitors. Secondly, the test scenarios are generated based on a definition of all possible inputs using a model checker, whereas we generate the test scenarios from a whitebox model of the system under test.

III. Methodology for Development and Testing of Complex Aircraft Systems

This section provides a description of our methodology in terms of the overall concept, the underlying metamodel and the envisaged process.

A. Concept

Our methodology combines monitor- and model-based testing to test the system model and the resulting system. Our aim is to achieve a high degree of reuse of artefacts from early development stages at later development stages and a high degree of automation throughout the process. Since we consider multiple levels of abstraction in our metamodel it is necessary to provide means which can verify a model at any abstraction level or the final product without the need for redeveloping the verification means for each verification stage. To this end, we use executable requirement monitors, which can be built very early on as soon as the first requirements are defined and which can be adapted easily for verifying the models or the product. Also, these monitors can be reused for testing different variants and/or design alternatives.

Figure 4 provides an overview of the main artefacts involved and their relations.

A requirement monitor is an exectuable model representing one requirement that, at any point in time, indicates the requirement violation status. The status should be enumerated with at least the following values:

- Not evaluated (default value), to indicate that the requirement has not been evaluated yet. Typically, this means that a necessary precondition has not been met yet.
- Not violated, to indicate that no violation has happened and implying that the requirement has been evaluated.
- Violated, to indicate a violation of the requirement and implying that the requirement has been evaluated.

This enumeration is referred to as three-valued semantics in [22] with the literals inconclusive, false and true respectively.

The monitor status can be obtained from a monitor at any point in time and can change between not evaluated, not violated and violated in any possible way. Following this approach, the status of the individual reuqirement monitors that are instantiated during one test can be used in aggregation to derive the test verdict. Removing the test verdict from the test cases will enable the reuse of test cases, that we now call test scenarios, for the verification against several requirements.

The task of converting the natural language statement into a formal language will require a correct interpretation of the requirement statement and the ability to translate the meaning into a model that expresses exactly the same. The general systematic way for deriving a monitor from natural language requirement is as follows:

1) Read the requirement statement
2) Identify properties that can be quantified either by explicit numbers or by logical conditions

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

3) Identify pre-conditions (if any), which must be satisfied before the requirement can be evaluated
4) Express when the requirement is violated and when not

Neither a particular design of the system nor scenarios are needed for formalizing a requirement. The resulting monitor can be used for the verification of any design alternative of the system using any scenario. Generally, the task of formalizing a requirement into a requirement monitors can be accomplished in many different ways using different formalisms. We decided to use SysML for the task because using the same notation for design and testing artefacts enables integrated development and testing without the need for additional tools or data converters.

We drive the tests using scenarios that we generate from the system models using MBT technology. Since we derive the test verdict from the requirement monitors independently from the system model we can use the scenarios derived from the system model to actually verify the system model as well as the final product.

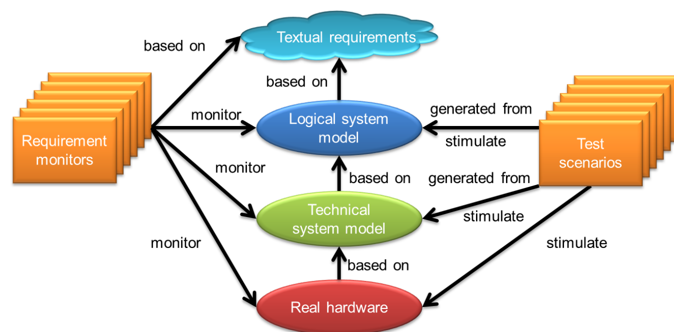


Fig. 4: Model-based testing using monitors

B. Meta-model

For our purpose, we extended the already established meta model for functional and systems architecture modeling [25] to allow a distinction between the functional, logical and the technical architecture of the system as depicted by Figure 5.

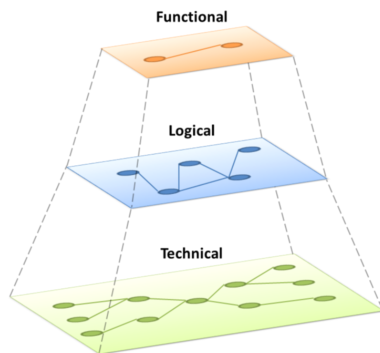


Fig. 5: Levels of abstraction

The main rationale for the distinction between these different layers is reusability. Between different aircraft programmes the functional architecture of a system is quite stable whereas the implementation can differ drastically. For a given aircraft programme the logical architecture is fixed quite early but different technical implementations might be considered and compared in trade studies. Ideally, we can now reuse the same functional architecture that is mature and proven and derive different logical and even more possible technical implementations that satisfy these functional needs.

The functional architecture, consisting of functions and data exchanges via functional dependencies is mapped to a logical system architecture, consisting of logical components that are instances of logical component classes and logical links between these components. This logical architecture can then in turn be mapped to the technical architecture of the system, which contains technical components, i.e., devices, and technical connectors, i.e., cables that connect the components. As can be seen from Figure 6 the relations between the elements in the different modelling layers allow a full traceability. This is crucial especially for maintaining the consistency of the models after changes.

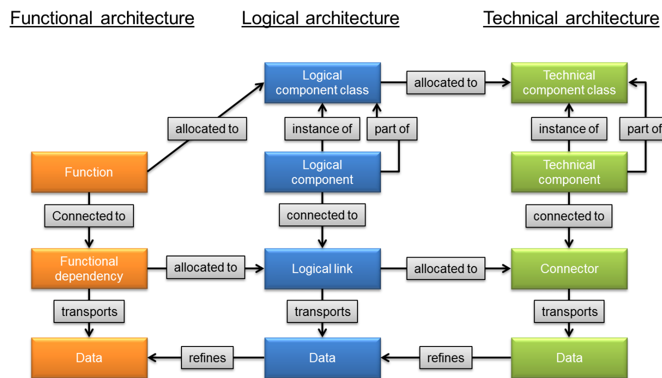


Fig. 6: Meta-model for current approach

While the modelling of the functional architecture in our approach is purely descriptive, the logical and the technical system architecture models are fully executable. Typically, the complexity of the models increases from the functional over the logical to the technical model. This is mainly due to two reasons: Firstly, when following this top down approach for systems modelling the level of abstraction decreases, which in turn increases the level of detail and complexity. Secondly, most aircraft systems require a certain degree of redundancy to abide by the safety constraints. A fact which is normally not considered during the functional analysis, only partly in the logical design but has the most impact on the technical architecture.

C. Process

The overall process underlying our methodology is straight forward and consists of the following steps:

1) Formalize requirements: create a violation monitor for each requirement
2) Build system models
3) Generate test scenarios from system models using MBT
4) Prepare the test environment: instantiate the monitors of the requirements that can be tested using the

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

available scenarios and connect them to the SuT (models or real hardware) appropriately
5) Execute tests: run all defined scenarios
6) Evaluate tests: aggregate the individual statuses of the requirement monitors that were active during a test to derive a test verdict

IV. Future Work

This section provides a couple of topics for current or future work for extending the approach described in this paper. Apart from extensions to the framework, we are also working on the application of the methodology for a concrete industrial-based use case to validate the framework.

A. Combination of model-based testing and model-based analysis

Dijkstra's famous aphorism holds that tests can only show the presence of errors not their absence [26]. Analysis techniques, e.g., model checking can be used to proof required characteristics of a system. Model-based analysis (MBA) and testing are complementary quality assurance techniques since static and dynamic analysis provide altogether different types of information: typically, static analysis provides general information about a model of the system while dynamic testing provides specific information about the system under test itself. Substantial quality and cost improvement can be obtained when they are systematically applied in combination.

One example for such a combination of MBT and MBA is the application of MBA in form of a model checker to improve the completeness of a test suite generated from a whitebox model using MBT as Figure 7 shows. The problem that is addressed by this method is that the automatic test scenario generator does not always achieve to generate a test suite with 100% coverage (coverage for this scenario means model/code coverage). At the moment, manual effort is required to complete a test suite to achieve 100% coverage. This manual effort can be replaced by the application of a model checker. If a test case generator manages to cover 95 out of 100 states of a model using test scenarios then we can write properties that check the reachability of the remaining five states. If the model checker manages to reach a state then the proof trace provided by the model checker can be directly added to the test suite as a new test scenario. If the model checker cannot find a solution for reaching a state then the model needs to be adapted.

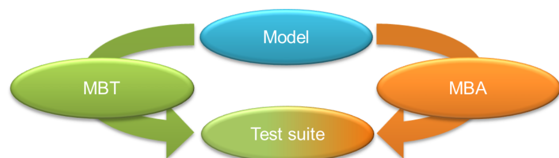


Fig. 7: Combination of model-based testing and model-based analysis

B. Combination of test scenarios obtained from different sources

Evluation of different MBT approaches and tools in the recent past, e.g., [27], [28] showed, that each tool has specific strengths and weaknesses and almost none of them can replace additional manual test scenario creation completely. If we use more than one test scenario generation approach and if we allow test scenario generation at different levels of abstraction as Figure 8 shows, then there is a high probability that the resulting test suite contains a high amount of redundant test scenarios. In order to test efficiently, especially when we are in the phase of hardware testing where a test run is much more expensive than a test run on a model, the redundancy in the test suite must be reduced to find an optimal test suite. Adaptation of previous work, e.g., [29], on that topic to our overall development and testing approach is currently being investigated.

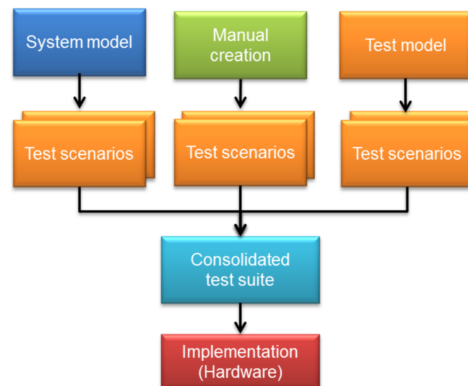


Fig. 8: Optimal test suite from different sources

C. Automated model-composition and results evaluation

The creation of models that integrate requirement monitors, a SuT system model and scenarios, i.e., the finding of suitable combinations of system model, scenarios and requirements, can be automated. Such a combined test model consists of one system model, which can be logical or technical, one scenario that can stimulate the design alternative and a set of requirements, which can be tested using the selected scenario. To automate the process further information is needed to evaluate the suitability of a combination of a test scenario and a design model, a test scenario and a requirement or a requirement and a system model. An approach for encoding this information and thereby enabling the automated composition of such test models is presented in [30]. Combining this approach with the one presented here is ongoing work. Additionally, running the tests, post-processing of the test results, and presenting the verification results appropriately can also be done in an automated fashion.

V. Conclusion

We presented a framework for an integrated development and testing approach of complex systems. The main driver behind this development was the need for more efficient testing. This was succesfully achieved by increasing the degree of reusability of engineering artefacts and automation of the testing process in the following way:

- Reusability
  - Explicit modelling of different architecture levels enables reuse of architectures.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

  - Requirement monitors can be reused for testing different architecture levels as well as the real hardware product.
  - Removing verdicts from test cases allows using the same test scenario for testing multiple requirements. Additionally, testing a requirement in different test scenarios increases the confidence in the conclusions drawn from the test results.
- Automation
  - Executable requirement monitors allow automated test verdict derivation.
  - Generation of scenarios using MBT.
  - Automated test execution of formal test scenarios.

The approach requires, as most model-based approaches, a frontloading of effort, a personnel shift and a different education of the involved people compared to the current state of practice. While evidence suggests that, through the high degree of reuse and automation, the effort for model-based testing is only slighly higher than the one for traditional testing [31] the adoption of the presented approach in an industrial environment nevertheless requires a rethinking of traditional roles and process steps.

ACKNOWLEDGMENT

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 269335 (ARTEMIS project MBAT) and from the German BMBF.

REFERENCES

[1] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” Software Testing, Verification and Reliability, vol. 22, no. 5, pp. 297–312, Aug 2012.

[2] V. Encontre, “Testing embedded systems: Do you have the GuTs for it,” IBM: http://www.ibm.com/developerworks/rational/library/459.html, Nov 2003.

[3] A. Helmerich et al., “Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area,” European Commission: http://www.artemis-austria.net/uploads/media/FAST_final-study-181105_en.pdf, Nov 2005.

[4] G. Laycock, The theory and practice of specification based testing. Department of Computer Science, University of Sheffield, 1993.

[5] P. Helle and W. Schamai, “Specification model-based testing in the avionic domain - Current status and future directions,” in Proc. Sixth Workshop on Model-Based Testing (MBT 2010), ser. ENTCS, vol. 264, no. 3, 2010, pp. 85 – 99.

[6] Object Management Group, “OMG Unified Modeling Language (OMG UML), v2.3,” 2010.

[7] ——, “OMG Systems Modeling Language (OMG SysML), v1.2,” 2008.

[8] J. Estefan, “Survey of Model-Based Systems Engineering (MBSE) methodologies,” INCOSE: http://www.incose.org/productspubs/pdf/techdata/mttc/mbse_methodology_survey_2008-0610_revb-jae2.pdf, 2008.

[9] P. Fritzson and V. Engelson, “Modelica - a unified object-oriented language for system modeling and simulation,” in Proc. European Conference on Object-Oriented Programming (ECOOP98). Springer, 1998, pp. 67–90.

[10] B. Selic, “The less well known UML,” in Formal Methods for Model-Driven Engineering, ser. LNCS, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer, 2012, vol. 7320, pp. 1–20.

[11] Object Management Group, “Semantics Of A Foundational Subset For Executable UML Models (FUML, v.1.0),” 2011.

[12] ——, “Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF), v1.0.1 beta,” 2013.

[13] D. Harel and H. Kugler, “The Rhapsody Semantics of Statecharts (or, on the executable core of the UML),” in Integration of Software Specification Techniques for Applications in Engineering, ser. LNCS, H. Ehrig et al., Eds. Springer, 2004, vol. 3147, pp. 325–354.

[14] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[15] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science). Springer, 2005.

[16] J. Zander-Nowicka, Model-based testing of real-time embedded systems in the automotive domain. Fraunhofer FOKUS, Berlin, 2009.

[17] T. Bauer, H. Eichler, A. Rennoch, and S. Wieczorek, Model-based Testing in Practice - Proc. 2nd Workshop on Model-based Testing in Practice (MoTiP 2009). University of Twente, The Netherlands, 2009.

[18] J. Boberg, “Early fault detection with model-based testing,” in Proc. 7th ACM SIGPLAN workshop on ERLANG. New York, NY, USA: ACM, 2008, pp. 9–20.

[19] H. Giese, G. Karsai, E. A. Lee, B. Rumpe, and B. Schatz, Model-Based Engineering of Embedded Real-Time Systems, ser. LNCS. Springer, 2010, vol. 6100.

[20] H. Le Guen, F. Valle, and A. Faucogney, Model-Based Testing - Automatic Generation of Test Cases Using the Markov Chain Model. Wiley, 2012, pp. 29–81.

[21] M. Lettrari, “Using abstractions for heuristic state space exploration of reactive object-oriented systems,” in FME 2003: Formal Methods. Springer, 2003, pp. 462–481.

[22] M. Leucker and C. Schallhart, “A brief account of runtime verification,” Journal of Logic and Algebraic Programming, vol. 78, no. 5, pp. 293–303, 2009.

[23] J. Levy, H. Saïdi, and T. E. Uribe, “Combining monitors for runtime system verification,” ENTCS, vol. 70, no. 4, pp. 112–127, 2002.

[24] C. Artho et al., “Combining test case generation and runtime verification,” Theoretical Computer Science, vol. 336, no. 2, pp. 209–234, 2005.

[25] P. Helle, “Automatic SysML-based safety analysis,” in Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems, ser. ACES-MB ’12. New York, NY, USA: ACM, 2012, pp. 19–24.

[26] E. W. Dijkstra, “The humble programmer,” Communications of the ACM, vol. 15, no. 10, pp. 859–866, 1972.

[27] M. Shafique and Y. Labiche, “A systematic review of model based testing tool support, Technical Report SCE-10-04,” Carleton University: http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-10-04.pdf, 2010.

[28] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, “A survey on model-based testing approaches: a systematic review,” in Proc. 1st Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech’07). ACM, 2007, pp. 31–36.

[29] G. Fraser and F. Wotawa, “Redundancy based test-suite reduction,” in Fundamental Approaches to Software Engineering, ser. Lecture Notes in Computer Science, M. Dwyer and A. Lopes, Eds. Springer, 2007, vol. 4422, pp. 291–305.

[30] W. Schamai, P. Fritzson, C. J. J. Paredis, and P. Helle, “ModelicaML value bindings for automated model composition,” in Proc. 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, ser. TMS/DEVS ’12. Society for Computer Simulation International, 2012, pp. 31:1–31:8.

[31] T. Bauer, F. Böhr, and R. Eschbach, “On MiL, HiL, statistical testing, reuse, and efforts,” in 1st Workshop on Model-based Testing in Practice (MoTiP 2008). Fraunhofer, 2008, pp. 31–40.

An Evaluation of Client-Side Dependencies of Search Engines by Load Testing

Emine Sefer, Sinem Aykanat
TUBITAK BILGEM YTKDM
Kocaeli, Turkey
emine.sefer@tubitak.gov.tr
sinem.aykanat@tubitak.gov.tr

Abstract— **Nowadays, web based large-scale systems, such as search engines, are widely used. The popularity of search engines created a new environment in which the applications need to be highly scalable due to the data tsunami generated by a huge load of requests. In this context, the main problem is to validate how far the web applications especially search engines can deal with the load generated by the clients. Load testing, in general, refers to the practice of accessing the system behavior under load. In this paper, we study on search engine performances' dependencies related to network bandwidth and Internet browsers in aspect of load testing. We observed that search engines' speed is dependent on Internet browsers and network bandwidth.**

Keywords-search engine; load testing; Internet browser; network bandwidth.

I. INTRODUCTION

Web-based applications are widely used nowadays because of their advantages, such as cross-platform without distribution and installation of software on thousands of clients, easy to be used and managed, etc. Therefore, web-based applications need to scale to thousands of concurrent users. To assure the quality of these systems, load testing is a required testing procedure in addition to conventional functional testing procedures, such as unit and integration testing [1].

Load testing, in general, refers to the practice of accessing the system behavior under load [1]. The load testing aims to identify and isolate system bottlenecks, tune application components, predict system scalability, and make judgments on system architecture or design, while performance models are used in analyzing the performance and scalability characteristics of the system under study [2].

In the literature, there are various load testing studies for web-based applications, using different technologies. One of these studies is conducted by A. Habul and E. Kurtovic. Their study presents a methodology for load testing an Ajax application [3]. Another study is about performance comparison between different web-based application architectures which are .NET and Java EE [4]. One of them is for peer-to-peer applications [5].

In this paper, we aim to present basic load testing approach for web applications with high intensity of use. Search engine applications are at the top of these applications.

From a user perspective, the client-side performance is more important than server performance. So, the client-side performance is considered for load testing in this study. Seeking client-side performance, response time, error rate, CPU usage and memory consumption are taken into consideration. These metrics are interpreted for two criterion, including bandwidth and browser for search engines by HP LoadRunner, which is a performance and load testing tool.

The paper starts by giving information on load testing and load testing metrics, then introduces testing environment. In fourth section, load testing results are given. Finally, the results obtained in this research are discussed.

II. LOAD TESTING

A. *Load Testing*

The analyze of the performance of the web-based system can be achieved using load testing and/or performance modeling approaches. Load testing is carried out to determine a system's behavior under both normal and expected peak load conditions. It helps to identify the maximum operating capacity of an application such as any bottlenecks and determine which element is causing degradation.

B. *Load Testing Metrics*

1) Response Time: Response time is a time defined by interval between client request and response from server. Response time is the key software performance metric for server-client applications.

2) Error Rate: Error Rate is the mathematical calculation that produces a ratio of unanswered requests to all requests. The percentage reflects how many responses are HTTP status codes indicating an error on the server, as well as any request that never gets a response. Error Rate is a significant metric because it measures "performance failure" in the application. It tells how many failed requests

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

are occurring at a particular point in time of your load test [7].

3) Client-side Resource Utilization (CPU Usage and Memory Consumption)

CPU usage is the amount of CPU time used by the Web Service while processing the requests and memory consumption is the amount of memory used by the Web Service while processing the requests.

III. TEST ENVIRONMENT

To measure dependencies of the search engine according to the intended use, a simple scenario was chosen. Scenario steps are provided below:

- Open browser
- Enter search engine address
- Type “wikipedia” query string
- Click search button
- Click “http://www.wikipedia.org/” address

The scenario contains three transactions that are opening search engine, searching and redirection to Wikipedia. It has been run for 1000 concurrent users for all cases. Concurrent users mean that all of users send their requests to the server at the same time.

This scenario was run for three search engines that are Google, Yandex and Bing according to two criteria, which are network bandwidth and Internet browsers. These Internet browsers are Google Chrome, Internet Explorer, Mozilla Firefox versions of which are supported HP LoadRunner. For the other case, two different network bandwidth values were selected: 1.5 Mbps (Asymmetric Digital Subscriber Line-ADSL) and 10 Mbps. Windows Performance Monitor application is used to measure CPU and RAM usage ratios.

Testing environment consists of a PC hardware that runs LoadRunner 11.5 testing tool. The technical characteristics of this PC are:

- Intel i5 CPU @3.2 GHz
- 4 GB RAM
- 64-bit operating system
- Windows 7 Professional

IV. EXPERIMENTAL RESULTS

In this study, in order to compare dependencies of search engines, load test scenario was run on two different cases. These are Internet browser and network bandwidth.

A. Internet Browser

The reason for choosing the browser is to understand whether speed of search engines depends on Internet browser or not.

Load test results of selected search engines are given in Table I, Table II and Table III for different browsers.

LoadRunner computed average response times of each transaction and we computed average response time that are given in Tables I-III as linear average of three transactions for each Internet browser.

TABLE I. BING PERFORMANCE COMPARISON

Bing	Search Engines		
	Google Chrome	**IE**	**Mozilla Firefox**
CPU	15%	13%	11%
RAM	38%	36%	40%
Average Response Time (s)	2,547	1,946	2,359
Error Rate	0,122	0,002	0,007

a. At 10 Mbps network bandwidth

TABLE II. YANDEX PERFORMANCE COMPARISON

Yandex	Search Engines		
	Google Chrome	**IE**	**Mozilla Firefox**
CPU	23%	15%	12%
RAM	38%	33%	33%
Average Response Time (s)	2,102	2,101	1,979
Error Rate	0,0003	0	0,0003

a. At 10 Mbps network bandwidth

TABLE III. GOOGLE PERFORMANCE COMPARISON

Google	Search Engines		
	Google Chrome	**IE**	**Mozilla Firefox**
CPU	22%	22%	20%
RAM	39%	39%	40%
Average Response Time (s)	3,368	3,417	3,143
Error Rate	0	0	0

a. At 10 Mbps network bandwidth

In Table I, load test results for Bing are shown. According to Table I:

- In terms of the use of PC resources, it is observed that Bing used CPU at least on Mozilla Firefox.
- In terms of the use of average response time, it is been observed that Bing was the fastest search engine running on IE.
- It is observed that errors in Bing are arised respectively due to the server and timeout period (LoadRunner timeout period: 120 s.).

In Table II, load test results for Yandex are shown. According to Table II:

- In terms of the use of PC resources, it is observed that Yandex used the least CPU on Mozilla Firefox.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-307-0

- In terms of the use of average response time, it is observed that Yandex was the fastest search engine running on Mozilla Firefox.

In Table III, load test results of Google are shown. According to Table III:

- In terms of the use of average response time, it is observed that Google was the fastest search engine running on Mozilla Firefox.

As a result, with regard to above statements, it can be considered that search engines' performance depend on Internet browser.

B. *Network Bandwitdh*

Network bandwidth used by the Internet user determines the speed of download and upload and is a parameter considered in the load test.

At first, widely used by the Internet users for network bandwidth ADSL (1.5 Mbps) is selected. Secondly, for putting forward dependence of search engines to network bandwidth, 10 Mbps, nearly ten times ADSL is selected.

The comparisons of 10 Mbps and ADSL for search engines are shown in the following tables.

We computed average response time that are given in Tables IV-VI as linear average of three transactions' average response times by LoadRunner for each Internet browser.

TABLE IV. BING PERFORMANCE COMPARISON

Bing	**Internet Browsers**					
	Chrome		**IE**		**Firefox**	
	ADSL	**10 Mbps**	**ADSL**	**10 Mbps**	**ADSL**	**10 Mbps**
CPU	14%	15%	14%	13%	14%	11%
RAM	39%	38%	36%	36%	39%	40%
Average Response Time (s)	3,117	2,547	2,921	1,946	3,04	2,359

TABLE V. YANDEX PERFORMANCE COMPARISON

Yandex	**Internet Browsers**					
	Chrome		**IE**		**Firefox**	
	ADSL	**10 Mbps**	**ADSL**	**10 Mbps**	**ADSL**	**10 Mbps**
CPU	25%	23%	28%	15%	26%	12%
RAM	38%	38%	44%	33%	36%	33%
Average Response Time (s)	2,405	2,102	2,416	2,101	2,35	1,979

TABLE VI. GOOGLE PERFORMANCE COMPARISON

Google	**Internet Browsers**					
	Chrome		**IE**		**Firefox**	
	ADSL	**10 Mbps**	**ADSL**	**10 Mbps**	**ADSL**	**10 Mbps**
CPU	22%	22%	21%	22%	22%	20%
RAM	40%	39%	30%	29%	40%	40%
Average Response Time (s)	3,7	3,368	3,466	3,417	3,731	3,143

In Table IV, it is observed that the decrease in network bandwidth only caused an increase in response time. In other words, we could say that Bing is slowed down when bandwidth is reduced.

In Table V, it is shown that when network bandwidth is reduced, CPU utilization of Yandex increased in IE and Firefox browsers. Also, it is observed that an increase in network bandwidth caused lower RAM usage by Yandex in IE and lower response times in all browsers.

In Table VI, it is observed that the decrease in network bandwidth only caused an increase in response times.

As a result, in terms of PC resource usage and speed, Yandex depends on network bandwidth. In terms of speed, Google and Bing browsers can be considered to be depended on network bandwidth.

C. *Comparing Search Engines in Point of Transactions*

The scenario contains three transactions that are opening search engine, searching and redirection to Wikipedia. The comparison of the transactions is given in Table VII.

We computed average response time that are given in Table VII as linear average of three Internet browsers' average response times by LoadRunner for each transactions.

The curves of variation of transactions' response time due to elapsed time for Google, Yandex and Bing are given in Fig. 1, Fig. 2, and Fig. 3, respectively.

TABLE VII. TRANSACTION PERFORMANCE COMPARISON

Average Response Time (s)	**Transactions**		
	Opening Search Engine	**Searching**	**Redirecting**
Bing	4,058	0,682	2,11
Yandex	2,995	1,467	1,720
Google	1,036	0,794	8,412

a. At 10 Mbps network bandwidth

According to opening search engine transaction, Google can be considered the fastest search engine. For searching transaction, it is observed that Bing and Google are faster than Yandex. By redirecting transaction, Google is said to be the slowest search engine. The reason for Google's slow redirecting is when user clicks the link, Google firstly send user their own servers to get information for their ranking algorithms and then provide the connection to selected link.

V. Conclusion and Future Work

In this paper, we aimed at presenting search engine performances' dependencies on network bandwidth and Internet browsers. We evaluated client-side performance of search engines for load testing.

As a result of load testing, it is observed that search engines' performance depend on Internet browser and Google is the least dependent on Internet browsers.

As network bandwidth increases, the utilization of PC resources by search engines decreases and speed of search engines increases as expected. However, usage of PC resources by Yandex increases. In this instance, Yandex is the most dependent on network bandwidth.

In future study, in addition to client-side load testing, it is planned to evaluate the behavior of the server during load testing.

Acknowledgment

The authors would like to thank Software Testing and Quality Evaluation Center (YTKDM in Turkish) of Scientific and Technological Research Council of Turkey (TUBITAK in Turkish) for funding this study.

References

[1] J. A. Meira, E. C. d. Almeida, Y. L. Traon, and G. Sunye, "Peer-to-peer Load Testing," Proc. IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), April 2012, pp. 642 – 647, doi: 10.1109/ICST.2012.153.

[2] O. Hamed and N. Kafri, "Performance Testing for Web Based Application Architectures (.NET vs. Java EE)," Proc. First International Conference on Networked Digital Technologies (NDT), July 2009, pp. 218 – 224 doi: 10.1109/NDT.2009.5272178 .

[3] A. Habul and E. Kurtovic, "Load Testing an AJAX Applications," Proc. 30th International Conferenece on Information Technology Interfaces (ITI 2008), June 2008, pp. 729-732, doi: 10.1109/ITI.2008.4588501.

[4] B. Beizer, Software System Testing and Quality Assurance, 2nd ed., Van Nostrand Reinhold, 1894, pp. 218-250.

[5] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic Identification of Load Testing Problems," Proc. IEEE International Conference on Software Maintenance (ICSM 2008), Sept.-Oct. 2008, pp. 307-316, doi: 10.1109/ICSM.2008.4658079.

[6] P. Yunming and X. Mingna, "Load Testing for Web Applications," Proc. 1st IEEE International Conference on Information Science and Engineeering (ICISE), Dec. 2009, pp. 2954-2957, doi: 10.1109/ICISE.2009.720.

[7] http://loadstorm.com/load-testing-metrics/, 2013.

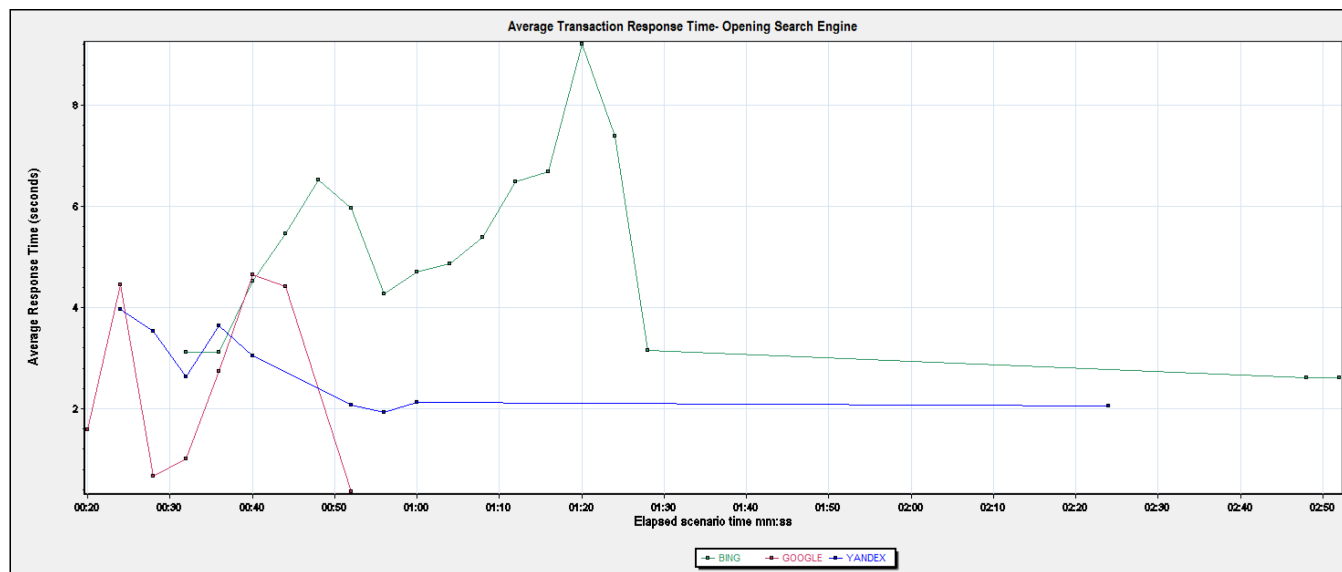


Figure 1. Opening search engine transaction response time.

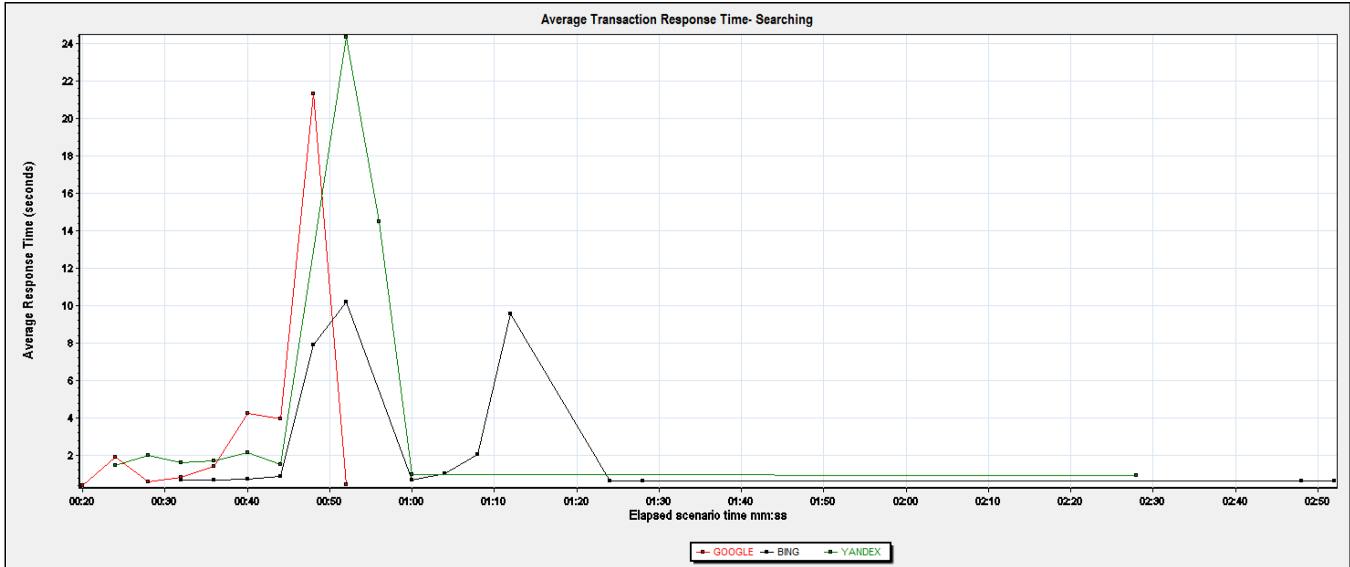


Figure 2. Searching transaction response time.

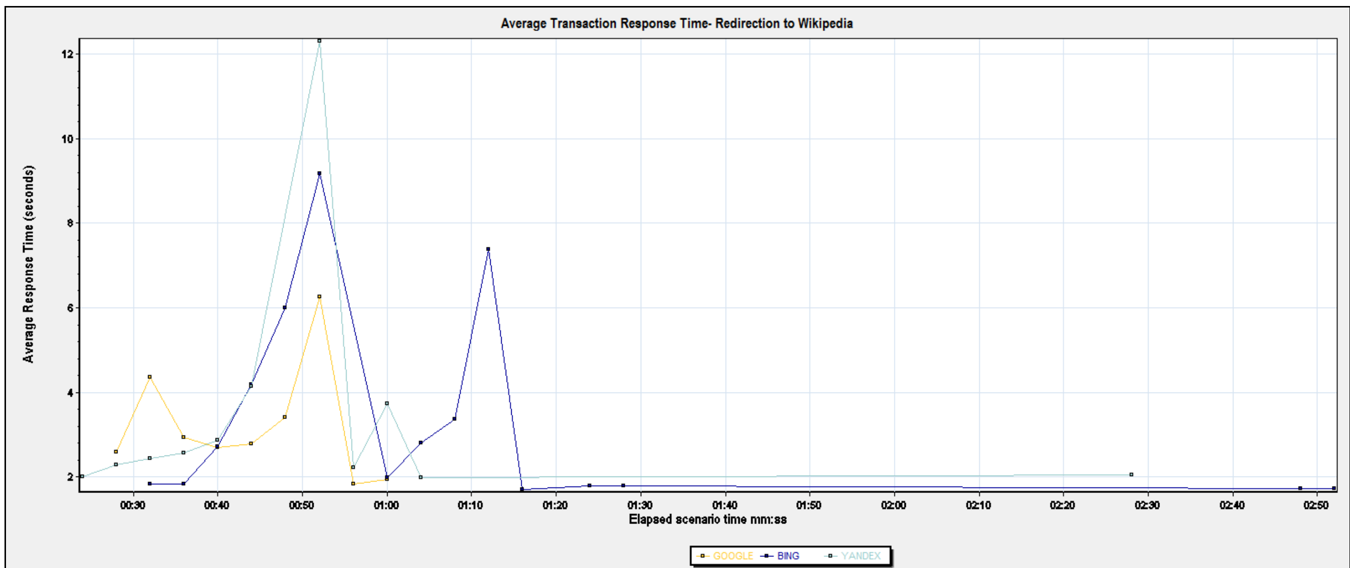


Figure 3. Redirection to Wikipedia transaction response time.

Compact Traceable Logging

I.S.W.B. Prasetya, Ales Šturala, Arie Middelkoop, Jurriaan Hage, Alexander B. Elyasov
Dept. of Inf. and Comp. Sciences, Utrecht Univ.
Utrecht, the Netherlands
Email: ales.sturala@hotmail.com, {A.Elyasov,S.W.B.Prasetya,J.Hage}@uu.nl, amiddelk@gmail.com

***Abstract*—Logging is a commonly employed technique to gather information about the dynamic behaviour of a program. The resulting logs can be analysed to derive statistics, infer models, to diagnose failures, and used for testing. Balancing the cost of logging (in terms of I/O time and disk usage) and the benefits of increasing logging details is a challenging task. In this paper, we present a source code transformation scheme that converts the given program with ordinary logging to enhance it with tracing information, and at the same time significantly reduces the size of the generated logs by applying a form of binary encoding. Decoders are generated to interpret the logs and establish how the executions that produced them flowed through relevant decision branches in the program. This paper describes the transformation for sizeable subset of sequential Java, including its complicated control structures. As a proof of concept, we have implemented a prototype.**

Keywords-software logging; logging; software tracing; tracing.

I. Introduction

With the rise of complexity of modern applications, it becomes impossible to fully anticipate their behavior prior to deployment. Many applications employ logging to provide diagnostic information about their dynamic behavior. It provides means to at least diagnose erroneous and unexpected behavior after the fact [1], [2], [3], and may even allow us to reconstruct the original execution. Other kinds of information can also be learned from logs, e.g., usage statistics, usage patterns [4], code coverage, profiling information, potential security breaches [5], and even behavior models and specifications [6], [7], [8]; the latter also imply that test-cases can thus be learned from logs as well.

Clearly, the information that can be extracted from logs ultimately depends on what is exactly logged. Ideally, given a program *P*, we want to generate *executable* and *reproducible* logs. Such a log can be interpreted (executed) to drive an execution of *P*, and will at least reproduce the same log. Being able to re-execute allows common debugging tools to be employed to diagnose the log. Unfortunately, most logs are not executable, let alone reproducible. A *traceable* log allows us to infer how *P*'s execution flowed when it produced the log. It is a weaker property (than executability), but the inferred information can still be useful for error diagnosis. However, the amount of information that must be stored in the logs to make them traceable is substantial, which ultimately slows down application execution, and leads to very large files.

Saving can be gained by not logging pieces of text, but instead encoding them as a (short) binary/bitstring that refers to the location in the program that produced it. Essentially, these bitstrings can be considered as indices into an array of log comments. Our contribution is as follows. We present a new binary encoding scheme. The basic idea is to choose this bitstring, such that so that it also encodes a fragment of control flow, and thus providing tracing information without costing us (many) additional bits –note that whatever representation we use, I/O overhead and disk usage can still be further reduced by post-compressing the resulting logs. Our approach works by transforming the statements of the original program, so that they produce the bitstrings that encode their control flows. The transformation also produces the corresponding decoders to interpret the produced bitstrings and reconstruct normal logs, enhanced with tracing information. The produced bitstring logs are significantly more compact; in our experiments, they are 2.5 - 40 times smaller than normal logs, while enhancing the logs (after decoding) by a factor of 3 - 11. The transformation-based approach also implies that our logging scheme is *transparent* to the programmers: they can in principle write their logging code as they always do, after which our transformation will extend them for free.

The problem is however non-trivial. Modern programming languages support a whole range of control constructs, e.g., switches and breaks, which trigger a non-standard execution flow that is hard to encode faithfully. We also have to deal with exceptions and external call-backs; they dynamically disrupt the normal execution flow, and thus disrupt the encoding. In this paper, we will address some of these constructs. As a proof of concept, we have built a prototype implementing the approach in Java. It covers more constructs than those mentioned here.

This paper is structured as follows. Section II defines what kind of tracing information we want to add. Section III explains the basic idea of our transformation-based approach. Section IV explains the transformation of basic control structures, such as **while**, but also **switch** and **break**. Exceptions and external call-backs are handled separately in Section V. Section VI shows some experiment results of our prototype. Section VII discusses related work. Section VIII concludes and mentions future work.

Copyright (c) IARIA, 2013. **ISBN: 978-1-61208-307-0**

II. Logging and Traceability

As an example, let us consider the program in Figure 1, consisting of just a single method, for sending some hello greetings. For logging, the primitive to use is $log(s, v)$; it writes the pair (s, v) to the log, where s is a descriptive string, and v is a value. The string s is static (its value does not depend on the program's state), whereas v is a dynamic value. The variant $log(s)$ can be used if we have no dynamic value to log.

```
1  void hello(int i) {
2    while(i>0){
3      if (decide(i)) {
4        log("Sending...") ;
5        send("hello world") ; }
6      if(even(i)) i=i/2 ;
7      else i-- ;
8    }
9    log("Done,i=",i) ;
10 }
```

Figure 1. The method hello.

An example of a log produced by the program is shown below:

```
Sending...Sending...Sending...Done,i=0
```

This log does not really tell us how the execution went through the program. Obviously, we can extend the logging to also trace the flow of control. But how can we do this without excessively increasing the overhead? Furthermore, we also notice that most characters in the above log actually belong to static strings. As suggested in Section I, they can be compacted, but can we combine compaction with tracing?

Let us first define what kind of tracing information we want to add. Let P be a single threaded target program. To help us defining our concepts, imagine G_P to be P's total control flow graph (CFG). If e is an edge in G, $m{\rightarrow}n$ denotes its pair of source and target.

We assume that every node in G_P has at most one call to *log*(..) (else we split the node). Such a node is called a *logging node*. An edge leading to such a node is called a *logging edge*. G_P's initial nodes are assumed to be non-logging.

Given an execution of P, its *history* is an imaginary log obtained as follows: (1) when the execution passes a logging edge e that calls $log(s, v)$, we append the tuple $(line(e), s, v)$ to the history, where $line(e)$ is the line number of e in P; (2) when the execution passes a non-logging edge f we append $line(f)$. Histories represent logging enhanced with maximum tracing information. Because the overhead to actually log them is usually too large, we will only log *partial* histories; but which edges are useful to be included?

A *sibling* of an edge $e = m{\rightarrow}n$ is another edge $m{\rightarrow}n'$ in G_P with the same source, but a different target. A *branch* is an edge that has siblings. A *path* in G_P is a sequence σ of edges, such that σ_i and σ_{i+1} are two connecting edges in G_P, with the same direction. A *full path* is a finite path that starts from an initial node of G_P and ends in a terminal node (or a node marked as an end-node if P is intended to run forever). An edge e *can reach* another edge f if there is a full path with $[e, f]$ as a subsequence. It *can avoid* f if there is a full path that passes e, but $[e, f]$ is not a subsequence of this path. A branch e is an *attractor* of f if it can reach f, and it has a sibling d that can avoid f. Conversely, d is a *distractor* of f if it can avoid f and it has a sibling that can reach f. So, an attractor always has at least one distractor as a sibling, and vice versa.

Let us, at least for now, decide to log this kind of edges:

Definition 1: An edge is *log-relevant* if it is either: (1) a logging edge, (2) an attractor of a logging-edge, or (3) a distractor of a logging-edge. □

So, a log-relevant edge is either itself a logging edge, or a branch that has been decisive towards reaching or not reaching a logging edge.

Note that passed attractors and distractors cannot in general be inferred back from the normal log (the one without tracing information). For example:

```
if(g){
  if(h) return ;
}
log(s)
```

The two else-branches above are two attractors towards reaching $log(s)$. When we see s in the log, we cannot from s itself tell which attractor was taken.

We also want to note that in a language with exceptions the above definition of log-relevance may become impractical; we will return to this issue later.

III. Transformation, the Basic Idea

We will first transform P to its so-called *tagged* version $\ominus P$, and deploy the latter. Rather than producing a normal log, $\ominus P$ produces a *binary trace* (or simply *trace*) which is a bitstring interspersed by $\langle v \rangle$ values. The transformation also constructs decoders, which are later used to decode/interpret the produced trace to reconstruct the corresponding normal log, enhanced with human-readable tracing information. The latter log is called *enhanced log*.

To minimize the amount of logged data, in $\ominus P$ we suppress the logging of static strings. So, calls to $log(s, v)$ in P are replaced by $log(v)$ in $\ominus P$, and calls to $log(s)$ are removed. The decoders will later reconstruct them from the given trace.

Definition 2: A *decision node* m is a node with outgoing branches. Such a node is *log-decisive* if it has an outgoing branch e that is an attractor of some logging-edge f (which also implies that e has a distractor as a sibling). □

For example, the guard of the **while** and the first **if** in Figure 1 are log-decisive. The guard of the 2nd **if** is not.

In $\ominus P$ we 'tag' all log-decisive nodes m. Some additional code is injected in m: (1) to assign a unique bitstring to each outgoing edge of m, and (2) writes this bitstring to the trace

when the edge is passed. So, if m only has two branches, then a single bit is sufficient to distinguish them. In Figure 2 we show the tagged version of the program in Figure 1.

```
void hello(int i) {
 while(tag(i>0)){
  if (tag(decide(i))) {
    send("hello world") ;
  if (even(i)) i=i/2 ; else i-- ;
 } ;
 log(i)
}
```

Figure 2. The tagged version of hello.

The function $tag(e)$ evaluates the Boolean expression e and returns the value; but as side effect it also writes the bit 1 to the log if e is true, and else 0. Effectively, this performs the tagging as meant above.

Notice that the static strings have been removed. Calls to $log(s, v)$ are converted to $log(v)$ that writes $\langle v \rangle$ to the trace. The execution that previously produced the log shown below Figure 1 now produces the following trace:

1111110⟨0⟩

The above string encodes which log-relevant edges were passed by the execution. The string is given to a decoder to produce the corresponding enhanced log. The decoder for hello is shown in Figure 3.

```
void dec_hello() {
  while (pop(3,9))
      if (pop(4,6))
          emit(5,"Sending...") ;
  emit(9,"Done,i=") }
```

Figure 3. *hello*'s decoder.

The original control flow of P is reflected by the decoder. We also include relevant line numbers information into the decoder. For each log-decisive node v in P (let's assume it only has two branches), the corresponding decision node v' in the decoder calls $pop(t, f)$. This pops the current bit b in the given trace. This bit tells us which branches of v that P took. The method $pop(t, f)$ simply returns the same bit, and this causes the decoder to follow the same branch at v'. The t and f are line numbers of v's branches in P; and they will be printed to the log accordingly by the decoder as tracing information.

Calls to $log(s, v)$ or $log(s)$ at line k in P are translated to $emit(k, s)$ in the decoder. When executed, it consumes the current $\langle v \rangle$ in the trace, if there is any; then writes either (s, v) or just s to the enhanced log, and decorating it by the line number k.

Other details of P are not carried over to the decoder.

Applying the decoder in Figure 3 to the above example trace produces the following enhanced log:

```
[Hello:2][Hello:3][Hello:4]@Sending...
[Hello:2][Hello:3][Hello:4]@Sending...
[Hello:2][Hello:3][Hello:4]@Sending...
[Hello:2][Hello:9]@Done,i=<0>
```

Once such a log is created, another tool could be written to actually step through the source code in an IDE, by following the tracing information.

Things can however get more complicated. The decoder above will actually be incorrect if $decide(i)$ also does logging, or if it throws an exception. Further adjustments are thus still needed. The next section describes the transformation in more details.

IV. Basic Transformations

More generally, P may have multiple methods. Only log-relevant methods need to be tagged, and for each such method a matching decoder should be generated. In the sequel, we will use the term 'statement' and 'expression' interchangeably. For a statement S, $\ominus S$ denotes its tagged version, and $\sharp S$ the corresponding decoder.

Recall that we want to tag log-decisive nodes. To decide which nodes are log-decisive, we do not actually want to construct G_P, since such a graph can be pretty large and is cumbersome to work with. We prefer to implement the transformation in a syntax directed way. We, therefore, decide the tagging based on the log-relevance of the corresponding statement, which is calculated syntactically:

Definition 3: A statement S is log-relevant, denoted by $S \in \mathbb{L}$, if it contains a call to $log(..)$ or a call to a log-relevant method. □.

Definition 4: A method m is log-relevant, denoted by $S \in \mathbb{L}$, if its body contains a call to $log(..)$ or a call to a another log-relevant method. □

The transformations are described by a set of transformation rules, which are denoted as the example in (1).

$$\text{VAL/VAR} \; \frac{v}{v \quad \epsilon} \quad v \text{ is a constant or variable} \tag{1}$$

The rule is named VAL/VAR, and specifies how constants and variables are transformed. The result of the transformation is a pair, as specified under the line; the left specifies the resulting tagged version, and the right one specifies the decoder. So, the rule above says that a constant or variable v is copied to the tagged program, but is removed (denoted by ϵ) from the decoder.

The rule to transform a whole method m is shown in (2). Only log-relevant m's need to be transformed. The decoder is another method named $dec_m()$. It has no formal parameter, though implicitly it takes a trace as its input.

$$\text{MDEF}\mathbb{L} \; \frac{\textbf{def } m(e_1...e_k)\ S}{\textbf{def } m(e_1...e_k)\ \ominus S \quad \textbf{def } dec_m()\ \sharp S} \quad m \in \mathbb{L} \tag{2}$$

Some of the rules to transform the body S are shown in Figure 4. They only deal with standard constructs; more complex constructs are discussed later.

The rule LOG causes the logging of the static string s to be suppressed in the tagged version. The decoder on the other hand, does $emit(l, s)$, where l is the line number of the call $log(s, v)$ in the original program.

$$\text{LOG}\ \frac{log(s,v)}{log(v)\quad emit(linenr,s)}$$

$$\text{IFN}\ \frac{\textbf{if}\ (e)\ S}{\textbf{if}\ (\ominus e)\ S\quad \sharp e}\quad S\notin\mathbb{L}$$

$$\text{IFL}\ \frac{\textbf{if}\ (e)\ S}{\textbf{if}\ (tag(\ominus e))\ \ominus S\quad \sharp e\ ;\ \textbf{if}\ (pop(l_t,l_f))\ \sharp S}\quad S\in\mathbb{L}$$

$$\text{ASSIGN}\ \frac{e_1 = e_2}{\ominus e_1 = \ \ominus e_2\quad \sharp e_1;\sharp e_2}\qquad \text{SEQ}\ \frac{S\,;T}{\ominus S\,;\ominus T\quad \sharp S\,;\sharp T}$$

$$\text{CONJN}\ \frac{e_1\ \&\&\ e_2}{\ominus e_1\ \&\&\ e_2\quad \sharp e_1}\quad e_2\notin\mathbb{L}$$

$$\text{CONJL}\ \frac{e_1\ \&\&\ e_2}{\ominus e_1\ \&\&\ \ominus e_2\quad \sharp e_1\ ;\ \textbf{if}\ (pop(l_1,l_2))\ \sharp e_2}\quad e_2\in\mathbb{L}$$

Figure 4. Transformation rules for standard constructs.

Rules IFL and IFN deal with the **if**(e) S structure (if-then). If S is log-relevant, then the then-branch is an attractor (of a logging node), so we have to tag the corresponding decision point (IFL). Else both branches of the if-then are neither an attractor nor distractor, and thus we should not tag the decision point (IFN). Note that the transformation is also recursively applied to e, since it may call a log-relevant method.

The rules for assignment (ASSIGN) and sequential composition (SEQ) are straightforward. Unary expressions $op\ e$, and binary expressions $e_1\ rop\ e_2$ where rop is an arithmetic or relational operator do not induce implicit control branching. Therefore, their rules are similar to assignments. Logical operators are more involved. The semantics of e_1 && e_2 in Java specifies that e_2 is not evaluated if e_1 turns out to be false (*short-circuiting*). This matters for the control-flow, in particular when e_2 is log-relevant. The rules CONJN and CONJL deal with this. Other operators with implicit branching, such as `||`, and `?:` can be treated analogously.

A. Method call and polymorphism

If a statement S calls a log-relevant method m, its decoder $\sharp S$ should also call the decoder of m. The rules to handle method calls are shown in (3) and (4).

$$\text{CALLL}\ \frac{\textbf{call}\ m(e_1...e_k)}{\textbf{call}\ m(\ominus e_1...\ominus e_k)\quad \sharp e_1;...;\ \sharp e_k;\ \textbf{call}\ dec_mD()}\quad m\in\mathbb{L} \tag{3}$$

$$\text{CALLN}\ \frac{\textbf{call}\ m(e_1...e_k)}{\textbf{call}\ m(\ominus e_1...\ominus e_k)\quad \sharp e_1;...;\ \sharp e_k}\quad m\notin\mathbb{L} \tag{4}$$

If the method has a receiver, we treat it as its first argument (so, we write $m(x,y)$ instead of $x.m(y)$).

However, to also handle polymorphism, these rules have to be extended. We will explain this with an example; consider the following program:

```
Person p = getPerson() ;
p.work() ;
```

Suppose *Person* has K number of subclasses that override *work*(). The decoder of the above statement will now have to figure out which variant of *dec_work* it has to call. The obvious case is when neither *Person*'s *work*() nor its variants are log-relevant. Then the call *p.work*() is *not* log-relevant, and we can use CALLN to handle it.

In other cases, the decision cannot in general be made statically. To handle this, we assign *Person*'s *work*() and its variants a unique variant-number *id* in the range $[0...K]$. The call *p.work*() is instrumented such that it checks at the run time which variant is called, and it writes *bits*(*id*) to the trace, where *id* the variant number and *bits*(*id*) returns its unsigned bits representation.

For example, suppose only *Student* overrides *Person*'s *work*(), and at least one of them is log-relevant. The decoder of the above statement first reads the encoded variant number and then, it interprets it to call the correct variant:

```
code = bits2num(pop(),pop()) ;
switch(code) {
  case 0 : Person.dec_work() ; break;
  case 1 : Student.dec_work(); break;
```

B. Jump-over

```
m() { if(a) x++ ;
      if(ok) return ;
      log("not ok") }
```

Figure 5. A method with a jump-over.

Consider the example in Figure 5. Since the then-parts of both conditionals are not log-relevant, the rule IFN will not tag the corresponding decision nodes, resulting in these tagged version and decoder:

```
m() { if(a) x++ ;
      if(ok) return ; }
```

```
dec_m() { emit(3,"not ok") ; }
```

This decoder is however incorrect, because it always produces "not ok", whereas the original m may not do so.

The problem arises from the rule for transforming sequential composition (SEQ): it assumes normal control flow. That is, any execution of $S;T$ always executes the T-part. Under this assumption, no edge in S can become an attractor or distractor of a logging-edge in T. Consequently, when transforming S we do not need to care about the log-relevance of T. However, statements like **return**, **break**, and **continue** (we call them *jump-over* statements) break this assumption. Note that not all jump-overs in S can actually be

distractors of a logging-edge in T, and calculating which of them are, introduces some overhead. We will instead redefine $S \in \mathbb{L}$ (Definition 3):

Definition 5: A statement S is *directly log-relevant*, denoted by $S \in \mathbb{L}_0$, if it contains a call to *log*(..), or a call to a log-relevant method (as in the old definition of "log-relevant" statement). □

Definition 6: A statement S is *log-relevant*, denoted by $S \in \mathbb{L}$, if either (1) $S \in \mathbb{L}_0$, or (2) it is a part of a log-relevant method *and* one of these holds:

2a) S contains a **return**.
2b) S is a part of a directly log-relevant **switch** and S contains a **break**.
2c) S is a part of a directly log-relevant loop and S contains a **break** or a **continue**.

□

With this new definition, the previous example will give the following tagged version and correct decoder:

```
m() { if(a) x++ ;
      if(tag(ok)) return ; }

dec_m() { if(pop(2,3) return ;
          emit(3,"not ok") ; }
```

Consider $S;T$ in a log-relevant method m. The new definition of log-relevance presumes that if an edge e in S is an attractor to a **return** node in S, it will also be a distractor of a logging edge in T. This is only true if T has a logging edge. We have a similar situation in **while**$(g)\{S;T\}$ if S contains a **break** or **continue**. This means that if T actually has no logging edge, we will end up logging some edges that are actually not log-relevant. But at least we do not miss one (so, the change above is safe).

Some rules need to be added as well; jump-overs change the flow of control, so they need to be copied to the decoder. These are in (5); *bc* is either **break** or **continue**.

$$\text{BRC}\ \frac{bc}{bc \quad bc} \qquad \text{RET}\ \frac{\textbf{return}\ e}{\textbf{return}\ \ominus e \quad \sharp e;\ \textbf{return}} \tag{5}$$

C. Switch statement

An **if** statement has two branches. So, when tagging it, one bit suffices to encode the choice between them. A **switch** statement may have N branches, with $N \geq 2$. This can be dealt with as follows. We overload the function $tag(e)$ to also take a bitstring as argument. It returns nothing, and writes the bitstring to the trace. We tag the k-th branch by $tag(bits(k))$ where $bits(k)$ is the bitstring (of length $n = {}^2log(N)$) that encodes the number k.

The absence of **break** in a switch branch requires a special treatment though. The control flow would implicitly "fall through" to the next branch. Consider this example:

```
switch (day) {
  case 6:   log("sat")
  case 7:   log("weekend") ; break ;
  default:  log("work")    ; break ;  }
```

If we just treat **switch** analogously to **if**, we would get the tagged version and decoder shown in Figures 6 and 7.

```
switch (day) {
  case 6:   tag(FF);
  case 7:   tag(FT); break ;
  default:  tag(TF); break ;  }
```

Figure 6. An incorrectly tagged fall-through switch.

```
int code = bits2num(pop(),pop()) ;
switch (code) {
  case 0: emit(2,"sat")       ;
  case 1: emit(3,"weekend") ; break ;
  case 2: emit(4,"work")    ; break ;  }
```

Figure 7. The decoder for a fall-through switch.

The decoder correctly consumes two bits. However, if $day = 6$ the tagged version will fall through and incorrectly produces four bits. The issue can be solved in two ways. One solution involves remembering the number of times a **switch** execution falls through; e.g., x times. The decoder must then consume $n+x*n$ bits, and discard the last $x*n$ bits. However, this involves quite a bit of bookkeeping. A much simpler solution, the one that we follow, is to simply duplicate the **switch**, in which the first of the duplicates executes a single call to *tag* for each case, and the second implements the original logic (but without the tags). This results in the tagged version in Figure 8, to be used with the decoder in Figure 7.

```
switch (day) {
  case 6:   tag(FF); break ;
  case 7:   tag(FT); break ;
  default:  tag(TF); break ; }

switch (day) {
  case 6:   ;
  case 7:   break ;
  default:  break ; }
```

Figure 8. Correctly tagged fall-through switch.

D. Loops

Consider first the following straight forward proposal to transform a **while**-loop:

$$\frac{\textbf{while}\ (e)\ S}{\textbf{while}\ (tag(\ominus e))\ \ominus S \qquad \begin{array}{l}\sharp e\ ;\\ \textbf{while}\ (pop(l_t, l_f))\\ \quad \{\ \sharp S;\ \sharp e\ \}\end{array}} \qquad \begin{array}{l} e \in \mathbb{L}\ \vee \\ S \in \mathbb{L}\end{array}$$

We can see it as the loop-version of the IFL rule. The reader may notice that unlike IFL now we also apply the transformation when only the guard is log-relevant. This is correct: the branch that goes into the loop's body would still be an attractor of the edge that goes from the body's end back to the guard. In contrast, in an **if**$(e)\ S$, the then-branch cannot be an attractor if S is not log-relevant.

The above rule is however incorrect if the loop contains a jump-over. Let us consider the example below; suppose $decide \in \mathbb{L}$.

```
while(decide(i)){ log("hi") ; continue ; }
```

The resulting tagged version and decoder:

```
while(tag(decide(i))) { continue ; }
```

```
dec_decide() ;
while(pop()){
  emit(1,"hi") ; continue ;
  dec_decide() ;
}
```

The decoder incorrectly produces the logs from *decide* exactly once, whereas the original loop can do this multiple times. The correct rule is shown in (6).

$$\text{LOOPL}\ \frac{\textbf{while}\ (e)\ S}{\begin{array}{ll}\textbf{while}\ (tag(\ominus e)) & \textbf{while}\ (true)\ \{\\ \quad \ominus S & \quad \sharp e\ ;\\ & \quad \textbf{if}\ (pop(l_t, l_f))\\ & \quad\quad \textbf{break}\ ;\\ & \quad \sharp S\ \}\end{array}}\quad \begin{array}{l} e\in\mathbb{L}\ \vee\\ S\in\mathbb{L}\end{array} \tag{6}$$

If the loop is not log-relevant, it is removed from the decoder, as shown in (7).

$$\text{LOOPN}\ \frac{\textbf{while}\ (e)\ S}{\textbf{while}\ (e)\ S \quad \epsilon}\quad e\notin\mathbb{L} \wedge S\notin\mathbb{L} \tag{7}$$

The transformations for do-while and for-loops are a bit more elaborate, although they follow the same general idea [9]. Note however, that simply treating **do** S **while**(e) as S; **while**(e) S does not work if S contains a jump-over.

E. Loop compaction

The above tagging scheme for loops is still problematical. Consider this example:

```
i=0 ;
while(i<n)
  if (i==999) log("special") ; i++ ;
}
```

Recall that we have required to also log distractors of log-relevant node (Def. 1). For the above loop, this means that every iteration always logs at least one bit, despite the fact that most, if not all, of its iterations will not actually produce any call to *log*(...). This can spam a very long bitstring, to eventually produce at most one static string in the enhanced log. To avoid this, we choose to discard the trace produced by an iteration if it does not actually pass any logging node. This is done by the following *loop compaction* algorithm.

There is a global stack *Lstack*. Elements of this stacks are pairs (i, z) where i is a so-called loop-ID, and z is a buffer where we can temporarily save a trace-fragment.

1) Every log-relevant loop H is instrumented such that when the loop is entered, a unique loop-ID *lid* is created. The id is created fresh every time the loop is entered, to distinguish between different invocations of the loop.
2) Whenever the guard g of H is evaluated, and it evaluates to true (so, a new iteration is about to start), a new buffer z is created, and the pair (lid, z) is pushed into *Lstack*. We maintain the following invariant:

   The traces buffered in Lstack never pass a logging node.

   So, when the execution of H cycles back to its guard, we can remove (lid, z) and all pairs above it from *Lstack*.
3) $tag(e)$ will write to the buffer at the top of *Lstack*, unless this stack is empty. Then it will write the bit directly into the trace file.
4) a call to $log(v)$ breaks the above invariant. So, we dump all buffers in *Lstack* (from bottom to top) to the trace file. And then clear the whole stack. From this point on *tag* and *log* will write directly to the trace file, until an iteration push a new (lid, z) into *Lstack*.
5) Because a loop may terminate through a jump-over, there may be some residual bits left in *Lstack*. The next logging node will cause these residual bits to be dumped into the trace file.

Using this scheme, the previous loop will produce a trace where it appears as if the loop immediately does the 999-th iteration, and then it exits.

F. Recursion

Recursions can cause a similar bits-spamming problem as loops. The above loop compaction algorithm exploits the property that every iteration of a loop returns to the loop's guard. On the other hand, a recursive function that calls itself multiple times (such as the example below) does not have a natural analogous of the 'loop-guard' concept. So, the above compaction scheme cannot be re-applied for recursions.

What we do is to wrap each recursive call with a dummy single iteration, e.g:

```
fib(n) {
  if (n==0) { return 0 ; }
  if (n==1) { return 1 ; }
  if (n>=10) { log("This may be too big") ; }
  int a,b ;
  int i=0 ; while(i<1) { a = fib(n-2) ; i++ ; }
      i=0 ; while(i<1) { b = fib(n-1) ; i++ ; }
  return a+b ;
}
```

This has the effect that if a recursive call to fib does not pass a logging node, it will not produce any tracing information either. Else, the execution from the first call to fib up to the logging node will be traced.

V. Exceptions

In most programming languages, many types of expressions can potentially throw an exception. In terms of control flow, such an expression introduces implicit branching, with one *implicit normal branch* that corresponds to the instruction's normal execution, and one or more *implicit*

exceptional branches that correspond to jumps to exception handlers. A typical program contains a lot of such implicit edges. Consider for example:

```
x++ ; a.x = x ; ... ; log("here") ;
```

The two assignments before `log` can throw an exception. The corresponding implicit exceptional branches are distractors of the *log*(..) statement, the assignments themselves become attractors. According to Definition 1, we have to log them as well. This would mean that a normal execution (one that does not throw any exception) that leads to a *log*(..) statement would generate additional tracing information belonging to *all* implicit distractors it passes; and there are many of them. This is too verbose! When an execution throws an exception, we are indeed usually interested in the corresponding tracing information. But when it does *not* throw any exception, we are much less interested in knowing which exceptions and which handlers it thus by-passed.

So, implicit normal branches are not going to be logged. We would want to log log-relevant implicit exceptional branches; but this is problematical for a different reason. To log these edges would mean that we have to instrument all subexpressions in *P* that can potentially throw an exception. The overhead would be unacceptable. To make it practical, we decide to only log the destination nodes; thus, the exception handlers. Thus, our logs can reveal which exceptions have been thrown, but not the specific subexpression that threw them.

```
log("Preparing") ;
try {
  prepare() ;
  try { x = receive() ; log("Received") ; }
  catch (ExcA a) { log("Ouch") ; }
  catch (ExcB b) { x=-1 ; }
}
catch (Exception e) { }
log("Done") ;
```

Figure 9. A try-catch statement.

Consider the example in Figure 9; assume that `prepare` and `receive` are not log-relevant. An exception handler *h* is logged if there is a log-relevant implicit exceptional edge *e* that goes to it (this can only be the case if *e* is a distractor or attractor of another log-relevant edge). This is the case for all handlers above. In general, in **try** *t* **catch** h_1 **catch** h_2..., if *t* or one of the h_k is log-relevant, then *all* handlers in the construct will be either an attractor or distractor, and thus have to be logged.

Now, consider first the following proposal of a tagged version of the statement in Figure 9; *logE*(*e*) is used to log a thrown exception:

```
try {
  prepare() ;
  try { x = receive() }
  catch(ExcA a) {logE(a); }
  catch(ExcB b) {logE(b); x=-1 ; }
}
catch (Exception e) { logE(e) ; }
```

Suppose the execution throws an *ExcA*. Indeed, this will be logged. The idea is to extend the decoder so that upon reading the logged exception it will *replay* it, and thus duplicate the original flow of control. However, just from the logged exception the decoder will not be able to infer whether the exception was thrown before the second **try** or in the second **try**, which is important to decide to which handler the control should flow.

To determine the right moment to replay the exception, we will do progress counting. A global variable τ : *int* is introduced for this purpose. The method *logE*(*e*) will now additionally log the value of τ; so in principle, now the decoder has the information to decide when it is the right moment to replay an exception.

The method *tick*() will be used to increase τ by one. We only need to *tick* when *P* passes points that matter for logging:

1) To distinguish if a logged exception is thrown before or inside a (log-relevant) **try-catch** structure, we tick just before we enter the structure.
2) To distinguish if a logged exception is thrown inside or after a **try/catch/finally**-section of a (log-relevant) **try-catch** structure, we tick when we reach the section's end.
3) For a similar reason, *tag*/*pop*, and *log*/*emit* implicitly call *tick*().

This results in the tagged version in shown Figure 10.

```
tick() ;
try {
  prepare();
  tick();
  try { x = receive()  ; tick(); }
  catch(ExcA a){logE(a); tick(); }
  catch(ExcB b){logE(b); x=-1; tick(); }
  tick(); }
catch (Exception e) { logE(e); tick() ; }
```

Figure 10. Correctly tagged try-catch statement.

```
tick() { check() ; tau++ ; }
```

Figure 11. Decoder's *tick*() also checks exception maturity

The counting of the progress should also be reflected in the decoder. That is, whenever the tagged version calls *tick*(), a *tick*() should be added at the corresponding place in the decoder. Furthermore, *pop* and *emit* implicitly calls *tick*() to match the same calls in *tag* and *log*. The decoder's version of *tick*() is slightly different, as shown in Figure 11. Before it increases its progress counter (τ), it checks whether the current item in the trace is a pair (e,t), representing an exception thrown at time *t*. If *t* is equal to the the current value of τ, we say that the exception *e has matured*. The decoder should then consume (e,t) from the trace and replay

e by throwing it. Else, (e,t) is not consumed an the decoder simply proceeds to its next statement.

The resulting decoder is shown in Figure 12.

```
emit(1,"Preparing") ;
tick();
try {
  tick() ;
  try{ emit(4,"Received"); tick(); }
  catch(ExcA a) { emit(5,"Ouch"); tick(); }
  catch(ExcB b) { tick(); }
  tick() ; }
emit(9,"Done") ;
catch(Exception e) { tick() ; }
```

Figure 12. Correct decoder for the try-catch statement.

The full transformation rule is shown in (8). It is for the case when the try-block is log-relevant. Else it will not be transformed.

$$\text{TRYCATCH}\quad \frac{\textbf{try}\ S\ \textbf{catch}(e)\ T\ \textbf{finally}\ U}{\begin{array}{ll} \begin{array}{l} tick(); \\ \textbf{try}\ \{\ \ominus S\ ;\ tick()\ \} \\ \textbf{catch}(e)\ \{ \\ \quad logE(e)\ ; \\ \quad \ominus T\ ;\ tick()\ \} \\ \textbf{finally}\ \{\ \ominus U\ ;\ tick()\ \} \end{array} & \begin{array}{l} tick()\ ; \\ \textbf{try}\ \{ \\ \quad \sharp S\ ;\ tick()\ \} \\ \textbf{catch}(e)\ \{ \\ \quad \sharp T\ ;\ tick()\ \} \\ \textbf{finally}\ \{ \\ \quad \sharp U\ ;\ tick()\ \} \end{array} \end{array}} \quad \begin{array}{l} S \in \mathbb{L}\ \vee \\ T \in \mathbb{L}\ \vee \\ U \in \mathbb{L} \end{array} \tag{8}$$

Additional handlers are transformed in the same way. However, when only the **finally**-part is log-relevant, we do not actually need to log the handlers (to add $logE$ there).

We also have to deal with uncaught exceptions. Such an exception will escape all handlers in the program (and then causes the program to crash). Such an exception is almost always log-relevant, so we need to log it as well, so that the decoder knows when to stop its current execution. To do so the body of the top-level entry point method (e.g., `main`) need to be wrapped by a fake **catch**-clause that catches any exception and rethrows it; the transformed version will then add the needed call to $logE$.

A. Untraced Call-backs

Most programs use standard libraries and other external libraries. When P calls an external method, this method may in turn call back to some method m in P. The latter may perform logging. During decoding, we have to ensure that dec_m is called. Normally, this is the responsibility of m's caller's decoder, ensuring that the original execution is faithfully imitated. The problem is that external libraries can not be assumed to have been exposed to our transformation; therefore, it has no decoder and thus, nobody will call dec_m. We call such a call back (to m) an *untraced call*.

It turns out that the solution we had to deal with exceptions (Section V) can be reused. Let us suppose it is possible to intercept untraced calls to m at the runtime. When such a call comes, we treat it as if an $Untraced_m$ exception has occurred, and log it (along with the current value of the progress counter). The name of the called method (m) is encoded in the name of the exception. The decoder will then be able to infer when this happened, and furthermore, it knows where to jump to proceed.

To be able to intercept untraced calls, we rename all log-relevant methods with fresh names; e.g., $m(x)$ to mz. In the target application, all calls to m are accordingly modified to calls to mz. Then we re-introduce the method $m(x)$ with the same signature, defined as in Figure 13.

```
m(x) {
  logE(new Untraced_m()) ;
  mz(x)  ; // call the original m
}
```

Figure 13. Wrapper to intercept untraced calls to m.

External methods that call to the original m will still call it with its old name, and thus will call the new m above. So, there we can code the logic of the interception, as shown above.

VI. Proof of Concept

As a proof of concept we built a prototype implementing the transformation discussed before. All Java control structures, except *labelled* break and continue, are handled We use Eclipse Java Development Tools (JDT) that allows Java source code to be analyzed and transformed at the abstract syntax tree (AST) level.

So far we assumed that the program P only has a single top-level entry point, e.g., its `main` method. This does not work for logging, e.g., a GUI application. Such an application is event driven: when a user interacts with it, e.g., by clicking on a button, it generates an 'event', and the corresponding event handler is executed. Each handler acts thus as a top-level entry point. In our implementation, it is possible to annotate multiple methods as top-level. A 'full execution' means an execution of a top-level method m, from its start to its end. Each full execution generates a separate trace file, e.g., named $log_m_timestamp.txt$. From such a name we can infer back to which decoder the file must be given. The trace is actually split into two log files: a $blog_m.txt$ file containing the pure bitstring part of the trace, and a $evlog_m.txt$ file containing dynamic values and thrown exceptions. This allows the bitstring to be stored more compactly.

To validate that our approach works, we try it on a number of examples, listed in Table I. LOCS is the total lines of code (comments and white lines are not counted). *TrT test suite* is a set of classes consisting of various logged statements that we use to test our implementation. *Reversi* is a small a GUI-based program implementing a game of the same name. This program has multiple top-level entry points. *Nanoxml* is an open source small XML parser. *Barred* is an open source file archiver. Except for the Trt test suite, these programs do not actually do any logging. We artificially add logging

statements, by converting most comments to calls to $log(s)$ or $log(s, v)$.

TABLE I SOME STATISTICS

	#class	*#methods*	LOCS
TrT test suite	33	59	629
Reversi	4	30	473
Nanoxml	25	285	3321
Barred	21	45	2075

We then run the examples on several sample inputs and compare the resulting logs and the logs that we would get if we do not apply our transformation. The results are shown in the table below.

TABLE II RESULTS

	org (KB)	*tr*	*enh*	*DVR*	*CR*	*ER*
TrT test suite	3.3	1.3	24.1	0.27	0.39	7.3
Reversi	29.6	7,5	87.6	0.09	0.16	3.0
Nanoxml	268	101	3054	0.18	0.38	11.4
Barred	29.2	1.4	111	0	0.05	3.8

Above, *tr* and *ehc* express the size, in KB, of the trace file and enhanced log. It is calculated by counting the number of characters; every character is counted as two bytes. The *org* is the size of the original logging fragments in *ehc*. $CR = tr/org$ is the obtained compaction ratio, and $ER = enh/org$ is an indication of the enhancement factor. So, the compaction factor of 0.38 for Nanoxml means that our generated trace is 0.38× smaller than what we would get from normal logging, whereas its enhancement factor of 11.4 means that after decoding we enrich the normal log with tracing information, roughly by 11.4×. The above way to calculate *ER* is indeed debatable. One may point out that we should instead compare the amount of raw information the logs carry. But this is also not very useful: the trace file can be thought as a minimal representation the raw information that the corresponding enhanced log embodies; but not even a tool can read a trace file without decoding it first. Enhanced logs produced by our implementation are intended to readable by human and parsable by tools. So they do contain some verbosity, but we did try to minimize it (e.g., we did not blow them up to HTML); so, comparing them to the size of the original logs seems reasonable. *DVR* is the ratio of the amount of logged dynamic values in the normal log. The above results indicate that higher *DVR* will decrease the compaction ratio, which is to be expected since, unlike static strings, dynamic values have to actually be logged.

We expected that the run time overhead would be less compared to normal logging, because we would have to do less I/O. However, in our experiments this does not turn out to be the case, as shown in the table below for the Nanoxml and Barred examples. *#calls* is the total number of calls to the *log* function, and *OV* is the resulting total time overhead in ms; *ovc* is the average overhead per number of call, and *ovs* is the average overhead per KB of logged data in the original log. These numbers indeed suggest that the overhead is quite small, but ideally those numbers should be negative. We believe that the implementation can still be improved by choosing more clever data structures in our implementation.

TABLE III TIME OVERHEAD

	#calls	*OV* (ms)	*ovc*	*ovs*
Nanoxml	5335	320	0.06	1.19
Barred	660	277	0.42	9.55

With respect to the conclusions suggested by the above results, the following are the threats to their validity.

1) Logging statements were artificially added; as said, by converting comments to calls to *log*. This resulted in quite intensive logging. A program with real logging may log with different intensity. In particular, when a program logs less frequently, it can be expected to also have more attractors/distractors between logging nodes. This affects *CR* negatively, but improves *ER*. *DVR* also matters; higher *DVR* affects *CR* negatively, without improving *ER*.
2) The amount of logging investigated was at most in the order of hundreds of KBs. In particular, we did not investigate large scale logging (e.g., in the order of GBs).

VII. RELATED WORK

Most work in logging has been focused on providing logging infrastructure for various software technologies. Many modern programming languages already come with logging libraries. These provide functions we can use to log messages. They often have a notion of 'logging level' to control the verbosity of the generated logs. There are also alternative libraries such as the Log4x family [10] that provide, e.g., improved APIs or improved performance. Apache Commons Logging offers a set of common logging APIs so that the implementation of the logger can be decoupled and replaced easily. Some SDKs, such as the Google Web Toolkit (GWT) for developing web applications may also come with its own logging library, specialized to the kinds of applications that they target. Most logs, e.g., web servers logs or OS logs, are semi structured [11], where for example types of events and their time stamps can be distinguished, but further information about them are often described in free style strings. Obviously, the more refined the structure is, the more viable they are for analysis. Some debuggers produce deeply structured logs [12]. FITTEST testing framework comes with PHP and Flash loggers that produce deeply structured logs [13], which are used to feed its model and oracle inference tools [8], [14].

Logging statements can be manually added into a program, or automatically inserted through program transformation. For example, this can be done by specifying the logging as a separate 'aspect', which is then weaved into the target program using an AOP tool like AspectJ [15], or using a special log instrumentation tool such as ABCi for Flash [16].

Or, we can use a generic program transformation tool such as Stratego [17]. Our approach can be seen as adding another layer of transformation. Our implementation is ad hoc, using JDT. In retrospect, using a tool like Stratego might have been a better choice, as it allows the transformation to be specified and composed more abstractly.

VIII. Conclusion & Future Work

We have presented a new log encoding scheme. The approach works by transforming the source code of the target program, to make its control flow statements to log bitstrings encoding their flows of control. The produced logs are significantly more compact than traditional logging, while at the same time, when decoded they enhances the resulting normal logs with substantial tracing information. To reconstruct the logs, decoders are needed; they are produced by the same transformation above.

A prototype implementing the approach has been built and tested on some real life programs. The results show 0.05 - 0.4 compaction ratio (2.5 - 20 times more compact), and 3 - 11 enhancement factor.

Future work. We want to investigate if the tracing information can be further enhanced, e.g., by logging runtime types of relevant objects. In theory, if such information is enumerable, the enumeration allows them to be compactly encoded, and thus the logging overhead is minimum.

We want to investigate if the logging scheme can be extended to multi-threaded programs. We believe that our scheme can be straightforwardly tweaked such that each thread writes to its own trace file (which is also good to maintain concurrency). However, when interpreting the resulting logs, we still want to infer what the temporal relations are between entries in the logs of different threads (e.g., does this entry e_1 from the thread T_1 occurs before e_2 from T_2?). Time stamping every bit in the trace is obvious not acceptable. However, we can log the time whenever T_1 manages to obtain a lock c, and when it releases it again. Because two threads cannot at the same time obtain the same lock, this will at least allow us to infer the happen-before relation between the threads.

We want to investigate if the logging scheme can be made more flexible by being able to *dynamically* turn on and off its tracing mode. Currently, it *always* produces tracing information, whether we want it or not. One situation where it would be useful to turn off tracing is when a loop/recursion spam too much tracing bits, despite the compaction scheme that we have applied. Turning tracing off is actually quite easy. However, the decoders relies on the tracing information to be able to correctly do their work. So, when a fragment of the trace is suppressed, the decoders have to be made smarter so that they can fill in the missing fragment on their own, so that they at least are able to continue decoding the rest of the trace.

Acknowledgment. This work is funded by the EU FITTEST project No. 257574.

References

[1] J. H. Andrews, "Testing using log file analysis: tools, methods, and issues," in Procd. 13th IEEE Int. Conf. on Automated Software Engineering, 1998, pp. 157–166.

[2] J. H. Andrews and Y. Zhang, "Broad-spectrum studies of log file analysis," in Procd. 22nd Int. Conf. on Software Engineering, 2000, pp. 105–114.

[3] H. Barringer, A. Groce, K. Havelund, and M. Smith, "Formal analysis of log files," AIAA Journal of Aerospace Computing, Information and Communications, 2010, pp. 365–390.

[4] S. Pachidi, "Software operation data mining: techniques to analyse how software operates in the field," Master's thesis, Dept. Inf. & Comp. Sciences, Utrecht Univ., 2011, IKU-3317153.

[5] K. Kowalski and M. Beheshti, "Improving Security Through Analysis of Log Files Intersections," I. J. Network Security, vol. 7, no. 1, 2008, pp. 24–30.

[6] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in Procd. of the 30th int. conf. on Software engineering. ACM, 2008, pp. 501–510.

[7] D. Tu, R. Chen, Z. Du, and Y. Liu, "A Method of Log File Analysis for Test Oracle," in Scalable Computing and Communications; Eighth International Conference on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM'09. International Conference on. IEEE, 2009, pp. 351–354.

[8] L. Mariani, A. Marchetto, C. D. Nguyen, P. Tonella, and A. I. Baars, "Revolution: Automatic evolution of mined specifications," in ISSRE, 2012, pp. 241–250.

[9] A. Sturala, "Record-based Logging," Master's thesis, Dept. Inf. & Comp. Sciences, Utrecht Univ., 2011, ICA-3324192.

[10] C. Gulcu, "Log4j delivers control over logging," Java World, 2000.

[11] A. Schuster, "Introducing the Microsoft Vista event log file format," digital investigation, vol. 4, 2007, pp. 65–72.

[12] M. Auguston, "A Program Behavior Model Based on Event Grammar and its Application for Debugging Automation," in AADEBUG, 2nd Int. Workshop on Automated and Algorithmic Debugging, 1995, pp. 277–291.

[13] I. S. W. B. Prasetya, A. Elyasov, A. Middelkoop, and J. Hage, "FITTEST log format (version 1.1)," Dept. of Inf. and Comp. Sciences, Utrecht Univ., Tech. Rep. UU-CS-2012-014, 2012.

[14] I. S. W. B. Prasetya, J. Hage, and A. Elyasov, "Using subcases to improve log-based oracles inference," Dept. of Inf. and Comp. Sciences, Utrecht University, Tech. Rep. UU-CS-2012-012, 2012.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in ECOOP'01, 2001, pp. 327–353.

[16] A. Middelkoop, A. Elyasov, and I. S. W. B. Prasetya, "Functional instrumentation of ActionScriptPrograms with ASIL," in Implementation and Application of Functional Languages, ser. LNCS, vol. 7257, 2011, pp. 1–16.

[17] E. Visser, "Stratego: A language for program transformation based on rewriting strategies," in Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings, ser. LNCS, vol. 2051, 2001, pp. 357–362.