VALID 2012

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

ISBN: 978-1-61208-233-2

November 18-23, 2012

Lisbon, Portugal

VALID 2012 Editors

Amir Alimohammad, San Diego State University, USA

Petre Dini, Concordia University, Canada / China Space Agency Center, China

VALID 2012

Forward

The Fourth International Conference on Advances in System Testing and Validation Lifecycle (VALID 2012), held on November 18-23, 2012 in Lisbon, Portugal, 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 2012 technical program committee as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and efforts to contribute to the VALID 2012. 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 2012 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 2012 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in system testing and validation.

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

VALID 2012 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

VALID 2012 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 2012 Industry Chairs
Abel Marrero, Bombardier Transportation Germany GmbH - Mannheim, Germany
Sebastian Wieczorek, SAP AG - Darmstadt, Germany
Eric Verhulst, Altreonic, Belgium

VALID 2012

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

VALID 2012 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 2012 Industry Chairs

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

VALID 2012 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, NRI Institute of Information Science and Technology - Bhopal, India
Mikey Browne, IBM, USA

Luca Cassano, University of Pisa, Italy
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
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
Robert Eschbach, ITK Engineering AG - Herxheim, Germany
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
Patrick Girard, LIRMM, France
Hans-Gerhard Gross, Delft University of Technology, The Netherlands
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
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
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
Omer Nguena-Timo, IRIT - ENSEEIHT/ Université de Toulouse, France
Roy Oberhauser, Aalen University, Germany
Johannes Oetsch, Vienna University of Technology, Austria
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
Andreas Raabe, fortiss - An-Institut der Technischen Universität München - Munich, Germany
Henrique Rebêlo, Federal University of Pernambuco, Brazil

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

MBPeT: A Model-Based Performance Testing Tool

Fredrik Abbors, Tanwir Ahmad, Dragoş Truşcan, Ivan Porres
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520, Turku, Finland
Email: {Fredrik.Abbors, Tanwir.Ahmad, Dragos.Truscan, Ivan.Porres}@abo.fi

***Abstract*—In recent years, cloud computing has become increasingly common. Verifying that applications deployed in the cloud meet their performance requirements is not simple. There are three different techniques for performance evaluation: analytical modeling, simulation, and measurement. While analytical modeling and simulation are good techniques for getting an early performance estimation, they rely on an abstract representation of the system and leave out details related for instance to the system configuration. Such details are problematic to model or simulate, however they can be the source of the bottlenecks in the deployed system. In this paper, we present a model-based performance testing tool that measures the performance on web applications and services using the measurement technique. The tool uses models to generate workload which is then applied to the system in real-time and it measures different performance indicators. The models are defined using probabilistic timed automata and they describe how different user types interact with the system. We describe how load is generated from the models and the features of the tool. The utility of the tool is demonstrated by applying to a WebDav case study.**

***Keywords*-Load Generation. Model-Based Performance Testing. Monitoring. Probabilistic Timed Automata. Models.**

I. INTRODUCTION

With the recent advancements in cloud computing, we constantly see more software applications being deployed on the web. This opens up a broader window to reach out to new users. As traffic increases, the overall quality of such applications becomes an even more important factor, since most of the processing is done on the server side. Evaluating that these kinds of systems meet the performance requirements is no longer a trivial task. High response times, technical issues, and display problems can ultimately have a negative impact on the customer satisfaction and ultimately the profitability of the company. As a result, effective performance testing tools and methods are essential for verifying that systems meet their performance requirements.

The idea behind performance testing is to validate the system under test in terms of its responsiveness, stability, and resource utilization when the system is put under certain synthetic workload in a controlled environment. The idea behind the synthetic workload [1] is that it should imitate the expected workload [2] as closely as possible, once the system is in operational use. Otherwise it is not possible to draw any reliable conclusions from the test results.

Jain [3] suggests three different techniques for performance evaluation: analytical modeling, simulation, and measurement. While analytical modeling and simulation are good techniques for getting early performance estimation, they rely on an abstract representation of the system and leave out details related to the system configuration. Such details are problematic to model or simulate, however, they can be the source of the bottlenecks in the system. With the measurement technique one has to wait until the system is ready for testing while with the two former techniques one can start testing while the system is being developed.

Traditionally, performance tests usually last for hours, or even days, and only test a predefined number of prerecorded scenarios that are executed in parallel against the system under test (SUT). The major drawback with this approach is that it certain inputs that the system will face might be left untested. Therefore, we suggest the use of models that describes how the virtual users (VUs) interact with the system and a probabilistic distribution between actions. The synthetic workload is then generated from these models by letting virtual users execute these models.

In this paper, we present a tool that evaluates the performance of a system. The main contribution of this work is that the load applied to the system is generated in real-time from models, specified using Probabilistic Timed Automata (PTA). A tool designed in-house is used to generate the load and monitor different performance indicators.

We use our tool to answer the following questions about the system under test:

- What are the values of different Key Performance Indicators (KPIs) of the system under a given load? For instance, what are the mean and max response times and throughput for a given number of concurrent users?
- How many concurrent users of given types does the system support before its KPIs degrade beyond a given threshold?

The rest of the paper is structured as follows: In Section II we discuss the related work. In Section III we give an overview of challenges with load generation and in Section IV we present our tool. Section VI presents a case study and a series of experiments using our approach. Finally, in Section VII, we present our conclusions and we discuss future work.

 ISBN: 978-1-61208-233-2

II. Related Work

There exist a plethora of commercial performance testing and load generation tools. However, most of them generate load from static scripts or pre-recorded scenarios that are scripted and executed in batches. In this section, we have focused our attention on tools that use models as input.

Denaro et al. [4] propose a tool for testing the performance of distributed software when the software is built mainly with middleware component technologies, i.e. J2EE or CORBA. The authors claim that most of the overall performance of such a system is determined by the use and configuration of the middleware (e.g. databases). The authors also note that the coupling between the middleware and the application architecture determines the actual performance. Based on architectural designs of an application the authors can derive application-specific performance tests that can be executed on the early available middleware platform that is used to build the application with. Their tool differs from ours in the sense that they target middleware components only and they make use of stubs for components that are not available during the testing phase.

Barna et al. [5] present a model-based testing tool that tests the performance of different transactional systems. The tool uses an iterative approach to find the workload stress vectors of a system. An adaptive tool framework drives the system along these stress vectors until a performance stress goal is reached. Their tool differs from ours in the sense that they use a model of the system instead of testing against the real system. The system is represented as a two layered queuing model and they use analytical techniques to find a workload mix that will saturate a system resource.

Another similar approach is presented by Shams et al. [6]. There, the authors have developed a tool that generates valid traces or a synthetic workload for inter-dependent requests typically found in sessions when using web applications. They describe an application model that captures the dependencies for such systems by using EFSMs. Their tool outputs traces that can be used in well known load generation tools like *httperf* [7]. Their approach differs from our in the sense that they focus on off-line trace generation while we apply the generated load on-line to the system.

Ruffo et al. [8] have developed a tool called *WALTy*. The tool that generates representative user behavior traces from a set of Customer Behavior Model Graphs (CBMG). The CBMGs are obtained from execution logs of the system and a modified version of *httperf* is used to generate traffic from these traces.

III. Load Generation Challenges

In performance testing, one of the main challenges is the load generation. The reason why load generation is such a challenge is that there are so many ways to get it wrong. For instance, important user types may not have been identified. These important user types might have a significant impact on the performance of the system. Another example is that the users that one is simulating during testing behave differently than users in the real world. This can lead to the fact that the generated load does not conform to the load that real users would put on the system. In other words, for load generation to be successful, one needs to be able to generate load that represent the real user load as closely as possible. Failure to do so, often leads to incorrect decisions regarding the performance of the system.

In real life, users need some time to reflect over the information that they have received. This is what usually is referred to as *think time*. The think time specifies how long the user normally waits before sending a new request to the system. Defining a think time for an action is not always as simple as it might seem. For example, to get really accurate results, one needs to consider the time it takes for a web page to be rendered in the client machine and the time it takes for the user to find a new action. Usually the think-time is different for different actions. Hence, in the load generation process, there need to be a way to define a think time value for each individual action.

Traditionally, load generation has been achieved by defining static scripts or pre-recorded scenarios that are run or played back in batches or certain quantities. Even if the scripts are somewhat parameterized, they do not behave like real life users would do. For performance testing, and especially load generation, to make sense one must allow the virtual user to behave as dynamically as real users.

IV. MBPeT Tool

In our approach towards model-based performance testing, we have developed a tool called *MBPeT*. The tool has essentially three high levels purposes: (1) to generate load according to input parameters and send it to the system, (2) to monitor the key performance indicators (KPIs) and other system resources, and (3) to present the results in a test report. The key performance indicators [9] or the KPIs are quantifiable values that one wants to measure and track. Example of typical KPIs are: response time, mean time between failure, number of concurrent users, throughput, etc.

MBPeT accepts as input a set of models expressed as probabilistic timed automata, the target number of virtual users, a ramp function, duration of the test session, and it will provide a test report describing the measured KPIs.

A. Performance Models

The behavior of virtual users is described with probabilistic timed automata (PTA) [10]. The PTA (see Figure 1) describes a set of locations and a set of transitions that take the automaton from one location to another. A transition can have four different labels: a clock zone, a probability value, an action, and a reset. The clock zone is an integer value describing discrete time. The clock zone specifies how long the PTA waits until firing a transition and, in our

 ISBN: 978-1-61208-233-2

case, it is the equivalent of the think time. In the figure below, this is represented with the variable X. It is, however, possible to define more than one clock variable. In case of a branch in the PTA, the transition that is taken is based on its probabilistic value. Consider location *6* in the PTA figure below. One can reach location *7* with a probability of *p5* or reach location *2* with the probability of *p4*. Upon taking a transition, the associated action is being executed against the system. Whenever the action is executed there is a possibility to reset the clock variable. In the PTA below this is represented with *X:= 0*. Every PTA has an end location, depicted by a double circle, which eventually will be reached.

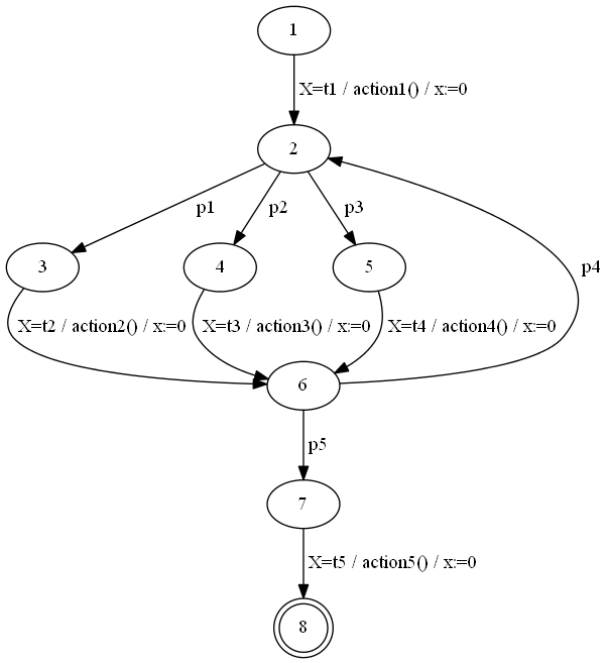


Figure 1. An example of a probabilistic times automata.

We believe that the PTA models are well suited for model-based performance testing and that the probability aspect that the PTA holds is good for describing dynamic user behavior, allowing us to include a certain level of randomness in the load generation process. This is important because we wanted the VUs to mimic real user behavior as closely as possible and real users do not follow static instructions. With the help of the probability values we can make it so that a certain action is more likely to be chosen over another action, whenever the VU encounters a choice in the PTA.

B. Tool architecture

The tool has a distributed architecture: a *master node* controls several *slave nodes* (see Figure 2.) The actual load generation is performed on the slave nodes. The master node just controls the load generation and initializes more slave nodes when needed. Each slave node is responsible for generating load for the VUs. The number of VUs a slave node can support is dependent on its capacity. In addition, each slave monitors its local resource utilization, collects KPIs for the system under test and reports the values to the master node.

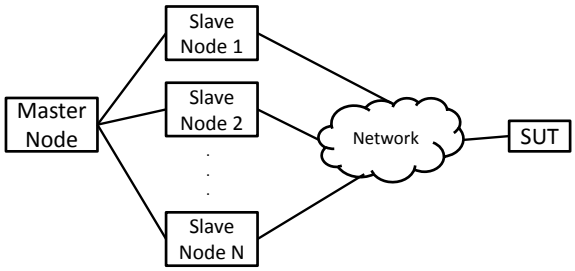


Figure 2. Master-Slave architecture for the MBPeT tool.

The internal architecture of the master node (Figure 3) includes the following components:

The **Model Parser** is responsible for reading the input models and building an internal representation of the model. In addition, it validates the models with respect to basic well-formness rules such as: all locations are connected, there is entry and an exit state, the sum of the probabilities of the transitions originating from a given node equals to 1, etc. We chose the *dot language* as a *plain text* representation for the PTAs. The reason for choosing the dot language is that we wanted to have a simple and lightweight way of representing models that both humans and machines can understand.

The **Core** module is the most important component of the master node. It takes care of reading the input parameters, initializing the test configuration by distributing relevant data to slave nodes, initiating load generation and collecting individual test reports from slaves. The test configuration contains information about the IP-addresses for the slave nodes and the master, the length of the test duration, a ramping function, and the number of concurrent users.

The master node uses two different **Test Databases**: *User DB* and *User-Resource Data Base*. The *User DB* contains data about the users, for instance user name and password, whereas the *User-Resource Data Base* contains information about the resources (documents, pictures, folders, etc) the users have on their own space on the server. The core module is responsible for initializing the data bases before the load generation begins.

The **Test Report Creator** module is in charge of producing an HTML test report once all the slave nodes have reported back to the master node all the gathered data from the test run. The report creator module aggregates the data and computes mean and max of the monitored values and for the specified KPIs.

 ISBN: 978-1-61208-233-2

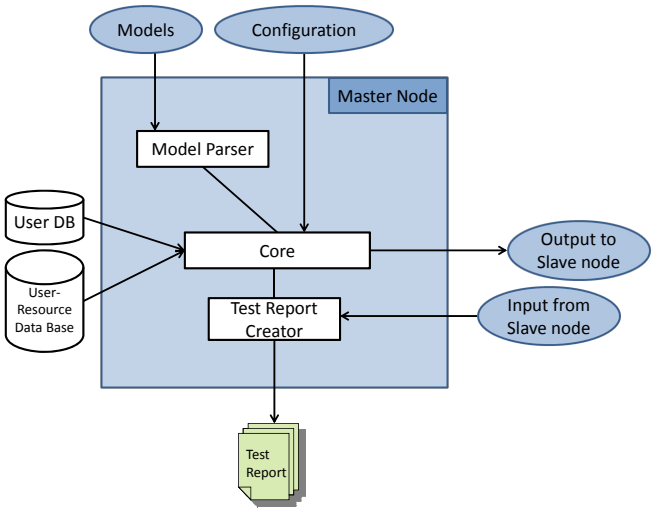


Figure 3. The structure of the master node.

The slave nodes consist of several modules (see Figure 4). The input received from the master node includes: the internal representations of the test models and specific test configuration. All the slave instances have identical configuration and implementation.

The **Load Generator** is in charge of generating load that is sent to the system under test. One instance of the PTA models is used by each VU to generate traces which results in sequences of actions based on specified probabilities and thin times that are sent to the SUT.

The models that we use contain abstract actions and can therefore, as such, not be directly used directly against the SUT. An **Adapter** module is used is to concretize every action into machine readableS format. For example, in the case of a HTTP-based system, a `login()` action needs to be implemented in the adapter code to be sent as a POST request over the HTTP protocol. A second example below shows how an *upload_file(image/jpg)* action on a WebDav server could be translated by an adapter:

```
PUT /webdav/user1/picture.jpg HTTP/1.1
Connection: Keep-Alive
Host: www.examplehost.com
Content-Type: image/jpg
```

A new adapter has to be implemented for each new SUT, however, it is possible to use different libraries in the adapter code to make the adaptation much easier. For example, in the case of the ”login()” action describe above, a standard HTTP library could be used to send the login to the SUT.

During the testing process each slave node monitors, via a **Resource Monitor**, the local resource utilization (CPU, memory, disk, and network) in order to make sure that the slave itself does not saturate and become a bottleneck in the configuration. If, for instance, the CPU utilization goes over a certain threshold, we can not guarantee anymore that the load is generated at the same rate as it should be. The slave node also monitors the response time for each action sent to the SUT and the error rate of these actions.

The **Reporter** module is in charge of putting the measured values together in an organized form and reports them back to the master node at the end of the testing process. The reporter is also responsible for notifying the master node if the local resource utilization threshold has been exceeded.

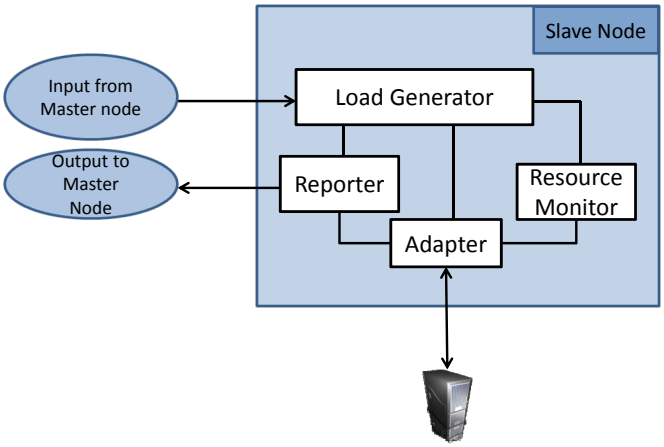


Figure 4. The structure of a slave node.

V. Performance Testing Process

In our approach, the testing process (see Figure 5) contains three distinct phases: test setup, load generation and test reporting.

A. Test Setup

The test setup phase takes care of initializing the test databases and the configuration of the slave nodes. This is done before the actual test run in order to avoid any negative impact on the bandwidth or resource utilization of the tester.

1) Test database initialization: One of the main challenges in performance testing is providing test data and configuring the system under test with a configuration as close as possible to the production environment [9]. As such, every time before starting the load generation phase, we configure the system under test and the tool with synthetic data using a populator script: on the system side, the script will automatically configure the web server with the given user configuration and if needed with the corresponding user spaces. On the MBPeT tool side the script will populate the user and test data databases with user credentials and corresponding information/files that the user will eventually upload to the server.

 ISBN: 978-1-61208-233-2

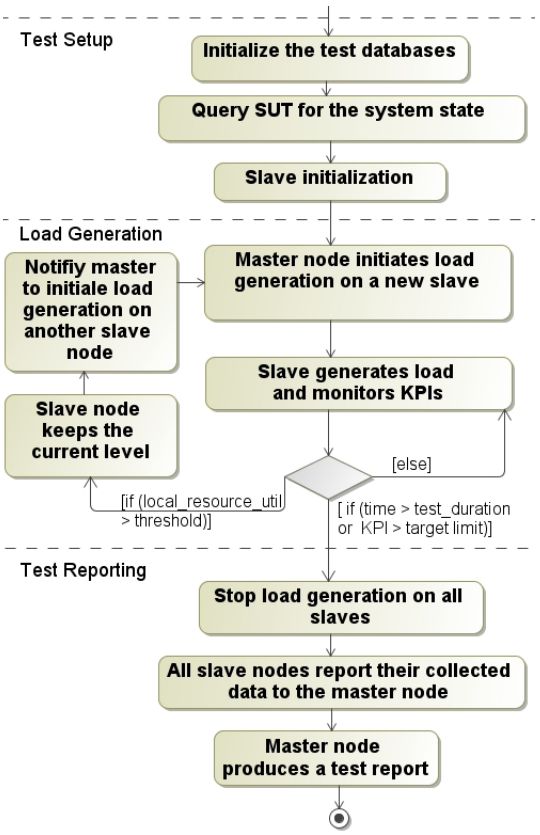


Figure 5. Activity diagram describing the load generation process.

2) System state: In certain cases, the current state of the SUT has to be captured before starting the test run. This is useful in stateful systems, in situations where the load generation should start from a given state of the SUT. For this, the master node queries the SUT for the user space resources, and stores the information in the URDB.

3) Slave initialization: When starting the testing process, the number of available slave nodes and their configuration (e.g., IP addresses) is provided to the master node. As mentioned previously, the master node will distribute the load incrementally on the slave nodes, one at the time, until the node saturates. However, all the available slave nodes are initialized with necessary test data and they are just idling.

B. Load generation

Different parameters of the testing process are provided as command line parameters. The tool can be used in *two* modes. The *first* use is to run with a certain target number of concurrent users. The tool will then slowly ramp up to the target number of users, run for the specified test duration and report back the aggregated values. The *second* use, and maybe the more interesting one, is to specify the target KPI value, for instance a target average response time, and let the tool find out the maximum number of concurrent users that the SUT can serve without exceeding the specified threshold.

To generate the load from the models, a few additional things have to be specified. First, one should input the models and a test configuration to the tool. Second, one needs to specify a stopping criterion for when the tool should stop generating load. This stopping criterion can be of two types: a time duration or a given threshold value. If a time duration is given, the tool will generate load based on the given models and target number of concurrent users, and stop generating load after the given amount of time has passed. If threshold values are given for a particular resource, e.g., the CPU, the tool will monitor that resource and ramp up the number of users until the threshold value is reached. All this information is specified in the configuration file. The load generation process will be discussed in more detail in Section VI

C. Test Reporting

When the specified test duration runs out or the tool detects that a certain threshold KPI value has been exceeded, the testing process is aborted and the test run is summarized. Consequently, each slave node reports back to the master node the data that it has collected during the test run. Based on the collected data the master node produces a test report of the test run.

The test report contains information such as, the duration of the test, number of generated users, amount of data sent to the system, response times for different actions, etc. The test report also shows diagrams of how various monitored values changed over time when the user amount was increased, e.g., response time, CPU, and resource utilization.

VI. Experiments

In this section, we demonstrate our tool by using it to test the performance of a Webdav [11] file server. Webdav (Web Distributed Authoring and Versioning) is an extension to the HTTP protocol and provides a framework for users to create, change, and move their documents and files stored on web servers. Webdav also maintains the file properties, e.g., author, modification date, file locking, etc. These features facilitate creation and modification of files and documents stored on web servers.

The SUT featured a Linux machine with 8-core CPU, 16GB of memory, 1Gb Ethernet, 7200 rpm hard drive, and Fedora 16 operating system. The file server ran a WebDav installation on top of an Apache web server. The system was configured for 1500 users, each with its own user space. The slave nodes that generated the load had the exact same configuration and were connected via a 1Gb Ethernet network to the SUT. In total we have used 3 slave nodes, but nothing prohibits us from extending this configuration.

By analyzing the Apache server logs of a previous WebDav installation with the AWStats [12] tool, we identified three user type: heavy, medium and light user, respectively, based on the average bandwidth each user type used for

 ISBN: 978-1-61208-233-2

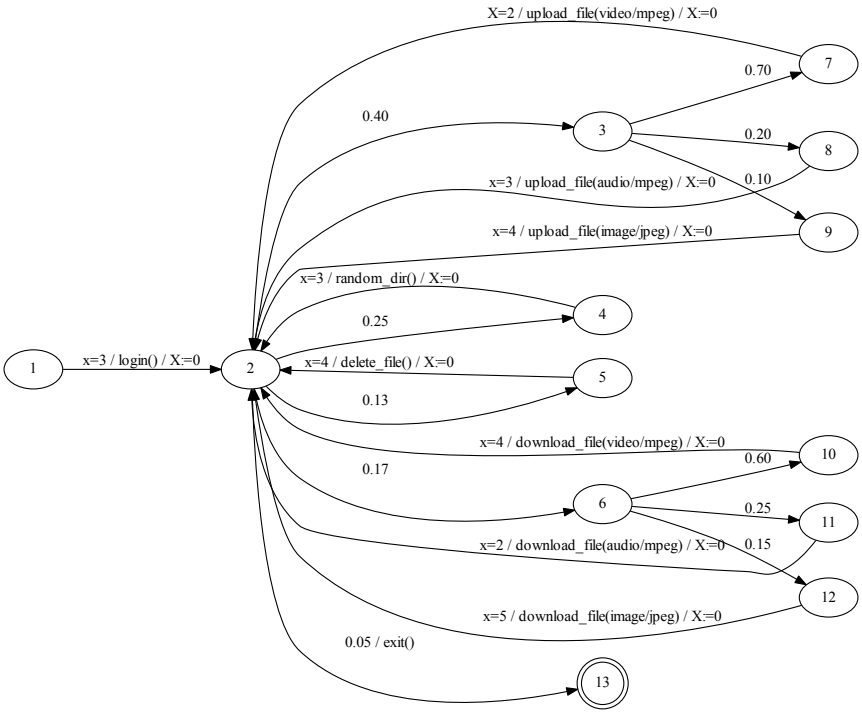


Figure 6. Probabilistic time automaton for a *heavy_user*.

transferring. The model depicting the distribution of these users is shown in Figure 7. We have also identified three types of file types (jpeg, mp3, avi) that the users usually transferred having on average file sizes of 3 MB, 5 MB, and 10 MB, respectively.

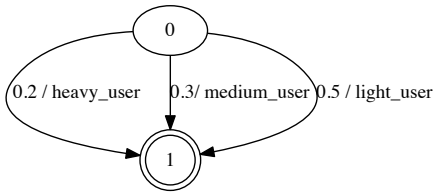


Figure 7. Distribution between different user types.

Figure 6 shows a PTA of the *heavy_user* type in terms of user actions, the probability for those actions, and the think time for each action. Eventually the user will find an *exit()* action and leave the system. Similar models were created for the *medium_user* and the *light_user* types. The PTA models for each user type can be completely different or be similar only varying in the distribution between actions. In our experiments we had the latter option.

The load generation process proceeds as follows: the master node takes as input the performance models, the test duration, the ramp function, the number of concurrent users and the target KPIs. The master node initiates load generation on the slaves in an incremental order. Each slave node monitors its local resource utilization and the KPIs of the SUT during the load generation. If the threshold for the local resource utilization on the slave node is exceeded, the slave node notifies the master node that it can not anymore generate new virtual users. The master then initiates load generation on another slave node. If the threshold of the measured KPI has been exceeded (in case a target KPI has been specified) or the test duration has ended the slave nodes notifies the master node and the load generation is stopped.

During the load generation on the slaves, the slave nodes execute the PTA models describing the user behavior as specified in Section IV-A in parallel processes. For each user the slave node starts a new process. The slave node then selects a user type if several are specified. The user type is selected based on probabilistic choice, see Figure 7. The virtual user then executes the PTA that belong to the selected user type inside its own process. Consider location *1* of the PTA in Figure 6. A possible execution of the PTA would be as follows: A virtual user waits until the clock variable X reaches *3* and then fires the transition. Upon firing the transition the action *login()* is sent to the adapter of the slave. The adapter creates a HTTP message, gets the user credential from the *User DB*, and sends the action to the SUT. In the adapter a timer is started to measure the response time. When the response is received it is checked in the adapter for the status code and the response time is stored. After that the clock variable *X* is reset to zero and

 ISBN: 978-1-61208-233-2

Table I
RESPONSE TIME MEASUREMENTS FOR USER ACTIONS WHEN RUNNING WITH 1000 CONCURRENT USERS.

	Light Users		Medium Users		Heavy Users	
Action	Average (sec)	Max (sec)	Average (sec)	Max (sec)	Average (sec)	Max (sec)
upload_file(video/mpeg)	82.3	133.0	81.5	133.5	85.1	133.3
upload_file(audio/mpeg)	158.3	217.4	143.7	214.3	126.2	210.5
upload_file(image/jpeg)	56.9	134.1	54.7	126.2	47.2	119.1
download_file(video/mpeg)	0.16	2.8	0.16	2.8	0.12	3.6
download_file(audio/mpeg)	0.15	3.0	0.18	3.2	0.18	3.0
download_file(image/jpeg)	0.13	3.1	0.15	3.7	0.16	1.4

the PTA moves from location *1* to location 2. In location *2* the transition to fire is based on the probabilistic values. For example, location 4 is reached with a probability of 0.25. In location *4* the transition is fired when the clock variable X reaches 3. The *random_dir()* action is sent through the adapter to the SUT. In this case, the adapter uses the *User-Resource Data Base* to select a folder for the user. Upon receiving the response the clock is reset and the PTA moves back to location 2. The process is repeated until the *exit()* action is fired and the end state is reached. The slave then chooses a new user type and the PTA corresponding to that user type is executed in a similar way. In a nutshell, every user runs independently of each other and decides for itself which actions to execute.

We have run two experiments with our tool, based on its two usage modes described in Section V-B

Experiment 1. In the first experiment, we wanted to answer the following question: *What are the mean and max response times of all actions when system is under the load of 1000 concurrent users?* We ran the test for 1 hour.

In this experiment we found out that the SUT had a bottleneck, namely the hard disk. Table I shows the average and max response times values for the actions and user types.

From the table one can see that the response times for the three upload actions are considerably higher that the ones for download. This is because a lot of data had to be written to the hard disk on the SUT, while the slave nodes simply discarded the data that the virtual users downloaded. Figure 8 shows the average response time plotted over time for the three upload actions for the *heavy_user type*.

Experiment 2. In the second experiment, we wanted to know *how many concurrent users of given types the system supports before the response time degrades beyond a given threshold?* The target threshold limits from the actions can be seen in Table II.

To figure this out, we had the tool to ramp up the number of user from 0 to 150 following the user type distribution in Figure 7 and the tool reported back when the measured response times exceeded any of the threshold values set for user actions specified in Table II. Figure 9 shows the average response times for the three upload actions plotted over time for the *medium_user* type when ramping up from 0 to 150 users. Similar graphs were created by the tool for the *light* and *heavy* user types. The test report also includes two tables for this experiment (see Table II and III).

Figure 8. Average response time for uploading picture (bottom), video (middle), and music (top) when running with 1000 concurrent users.

Table II shows the time and number of users at which the threshold value for individual actions was exceeded. Table III shows the average and max response times for individual action over the entire test durations. The tool reported that the average and max response times were exceeded for all of the three upload actions. However, the response time for *upload_file(audio/mpeg)* for the *medium_user* type went over the set threshold of 3.5 seconds at 8 minutes and 44 seconds (524 seconds) into the test run. The tool was then testing with 74 concurrent users. The distribution between user types was the following: 50% *light_users*, 28% *medium_users*, and 22% *heavy_users*.

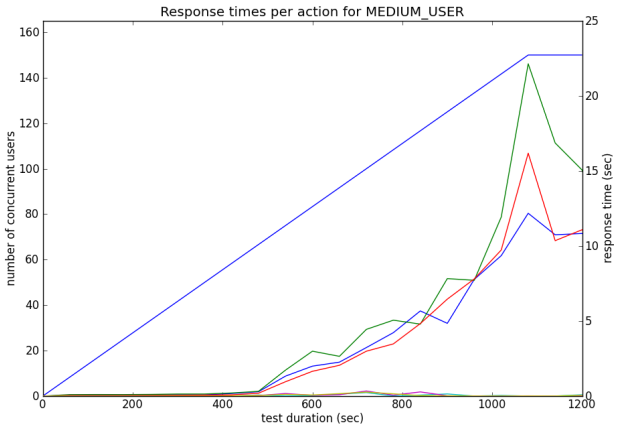


Figure 9. Average response times for uploading video (bottom), picture (middle), and music (top) when ramping up from 0 to 150 concurrent users.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a model-based performance testing tool that uses probabilistic models to generate

Table II
TIME AND NUMBER OF USERS AT WHICH THE THRESHOLD VALUE WAS EXCEEDED WHEN RAMPING UP FORM 0 TO 150 USERS

Action	Target Response Time		Light Users (50 %)		Medium Users (28 %)		Heavy Users (22%)		Verdict
	Average (sec)	Max (sec)	Average (users)	Max (users)	Average (users)	Max (users)	Average (users)	Max (users)	Pass/Fail
upload_file(video/mpeg)	4.5	12	78 (555.0 sec)	94 (669.0 sec)	86 (613.0 sec)	94 (670.0 sec)	78 (558.0 sec)	112 (801.0 sec)	Failed
upload_file(audio/mpeg)	3.5	10	76 (543.0 sec)	94 (670.0 sec)	74 (524.0 sec)	94 (670.0 sec)	74 (528.0 sec)	94 (671.0 sec)	Failed
upload_file(image/jpeg)	2.5	8	77 (545.0 sec)	80 (572.0 sec)	74 (524.0 sec)	80 (572.0 sec)	78 (555.0 sec)	101 (719.0 sec)	Failed
download_file(video/mpeg)	4.5	12	Passed	Passed	Passed	Passed	Passed	Passed	Passed
download_file(audio/mpeg)	3.5	10	Passed	Passed	Passed	Passed	Passed	Passed	Passed
download_file(image/jpeg)	2.5	8	Passed	Passed	Passed	Passed	Passed	Passed	Passed

Table III
RESPONSE TIMES WHEN RAMPING UP USERS FOR 0 TO 150 USERS

	Light Users		Medium Users		Heavy Users	
Action	Average (sec)	Max (sec)	Average (sec)	Max (sec)	Average (sec)	Max (sec)
upload_file(video/mpeg)	4.78	79.17	4.08	40.79	4.72	64.09
upload_file(audio/mpeg)	5.34	92.79	5.57	90.20	6.60	92.79
upload_file(image/jpeg)	4.22	93.44	4.25	93.04	5.04	87.84
download_file(video/mpeg)	0.05	1.98	0.05	1.57	0.04	1.57
download_file(audio/mpeg)	0.04	1.44	0.07	2.11	0.04	1.33
download_file(image/jpeg)	0.05	2.04	0.05	2.10	0.06	1.99

synthetic workload which is applied to the system in real-time. The models are based on the Probabilistic Timed Automata, and include statistical information that describes the distribution between different actions and think time. The tool has a scalable distributed architecture with a master node that controls several slave nodes. The slave nodes monitor the target KPIs and the local resource utilization, and after the test duration has ended the monitored values are sent to the master node which produces a test report.

We have described how load is generated from the PTA models and we have also discussed the most important features of the tool. We demonstrated the utility of the tool on a WebDav case study. We use our tool to answer the two questions about the system under test: What are the values of different KPIs when the system is under a particular load and how many users of given types does the system support before its KPIs degrade beyond a given threshold?

In the future, we will focus on the creation of the models and try to optimize the algorithm for load generation even further. We will strive to have a more formal approach on how we go from requirements to model. Also we will look further into load generation, for instance, develop methods to specify a minimum number of user action that has to be fulfilled (trace lengths) before the user can exit the system.

We are currently performing larger scale experiments to evaluate the capabilities of the tool against existing tools like, *JMeter* [13] and *httperf* [7]. Further, we plan to add target KPI values, for instance response time and performance requirements, in the models. By doing that, we can have performance requirements and address target response time values for individual actions.

REFERENCES

[1] D. Ferrari, "On the foundations of artificial workload design," in *Proceedings of the 1984 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '84. New York, NY, USA: ACM, 1984, pp. 8–14.

[2] J. Shaw, "Web Application Performance Testing – a Case Study of an On-line Learning Application," *BT Technology Journal*, vol. 18, no. 2, pp. 79–86, Apr. 2000.

[3] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling (Book Review)," *SIGMETRICS Performance Evaluation Review*, vol. 19, no. 2, pp. 5–11, 1991.

[4] G. Denaro, A. Polini, and W. Emmerich, "Early performance testing of distributed software applications," in *Proceedings of the 4th international workshop on Software and performance*, ser. WOSP '04. New York, NY, USA: ACM, 2004, pp. 94–103.

[5] C. Barna, M. Litoiu, and H. Ghanbari, "Model-based performance testing (NIER track)," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 872–875.

[6] M. Shams, D. Krishnamurthy, and B. Far, "A model-based approach for testing the performance of web applications," in *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*. New York, NY, USA: ACM, 2006, pp. 54–61.

[7] Hewlett-Packard, "httperf," http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt, retrieved: October, 2012.

[8] G. Ruffo, R. Schifanella, M. Sereno, and R. Politi, "WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance," *Network Computing and Applications, IEEE International Symposium on*, vol. 0, pp. 77–86, 2004.

[9] D. A. Menasce and V. Almeida, *Capacity Planning for Web Services: metrics, models, and methods*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[10] M. Jurdziński, M. Kwiatkowska, G. Norman, and A. Trivedi, "Concavely-Priced Probabilistic Timed Automata," in *Proc. 20th International Conference on Concurrency Theory (CONCUR'09)*, ser. LNCS, M. Bravetti and G. Zavattaro, Eds., vol. 5710. Springer, 2009, pp. 415–430.

[11] *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*, http://www.webdav.org/specs/rfc4918.pdf, Network Working Group pdf, retrieved: October, 2012.

[12] AWStats, http://awstats.sourceforge.net/, retrieved: October, 2012.

[13] The Apache Software Foundation, "Apache JMeter," http://jmeter.apache.org/, retrieved: October, 2012.

 ISBN: 978-1-61208-233-2

Cost-Aware Combinatorial Interaction Testing

Gulsen Demiroz and Cemal Yilmaz
Faculty of Engineering and Natural Sciences
Sabanci University, Istanbul 34956, Turkey
Email: {gulsend,cyilmaz}@sabanciuniv.edu

***Abstract*—The configuration spaces of modern software systems are often too large to test exhaustively. Combinatorial interaction testing approaches, such as covering arrays, systematically sample the configuration space and test only the selected configurations. Traditional t-way covering arrays aim to cover all t-way combinations of option settings in a minimum number of configurations. By doing so, they assume that the testing cost of a configuration is the same for all configurations. In this work, we however argue that, in practice, the actual testing cost may differ from one configuration to another and that accounting for these differences can improve the cost-effectiveness of covering arrays. We first introduce a novel combinatorial object, called a *cost-aware covering array*. A t-way cost-aware covering array is a t-way covering array that minimizes a given cost function. We then provide a framework for defining the cost function. Finally, we present an algorithm to compute cost-aware covering arrays for a simple, yet important scenario, and empirically evaluate the cost-effectiveness of the proposed approach. The results of our empirical studies suggest that cost-aware covering arrays, depending on the configuration space model used, can greatly reduce the actual cost of testing compared to traditional covering arrays.**

***Keywords*-Software quality assurance, combinatorial interaction testing, covering arrays.**

I. INTRODUCTION

The configuration spaces of configurable software systems are often too large to test exhaustively. The number of possible configurations is often far beyond the available resources to test the entire configuration space in a timely manner, e.g., for regression testing.

Combinatorial interaction testing (CIT) approaches take as input a configuration space model. The model includes a set of configuration options, each of which can take on a small number of option settings. As not all configurations may be valid, the model can also include some system-wide inter-option constraints. In the context of this work, an inter-option constraint is a constraint that implicitly or explicitly invalidates some combinations of option settings. In effect, the configuration space model implicitly defines a set of valid ways the software under test can be configured.

CIT approaches systematically sample the valid configuration space and test only the selected configurations. The sampling is carried out by computing a combinatorial object, called a *covering array*. Given a configuration space model, a t-way covering array is a set of configurations, in which each possible combination of option settings for every combination of t options appears at least once [6].

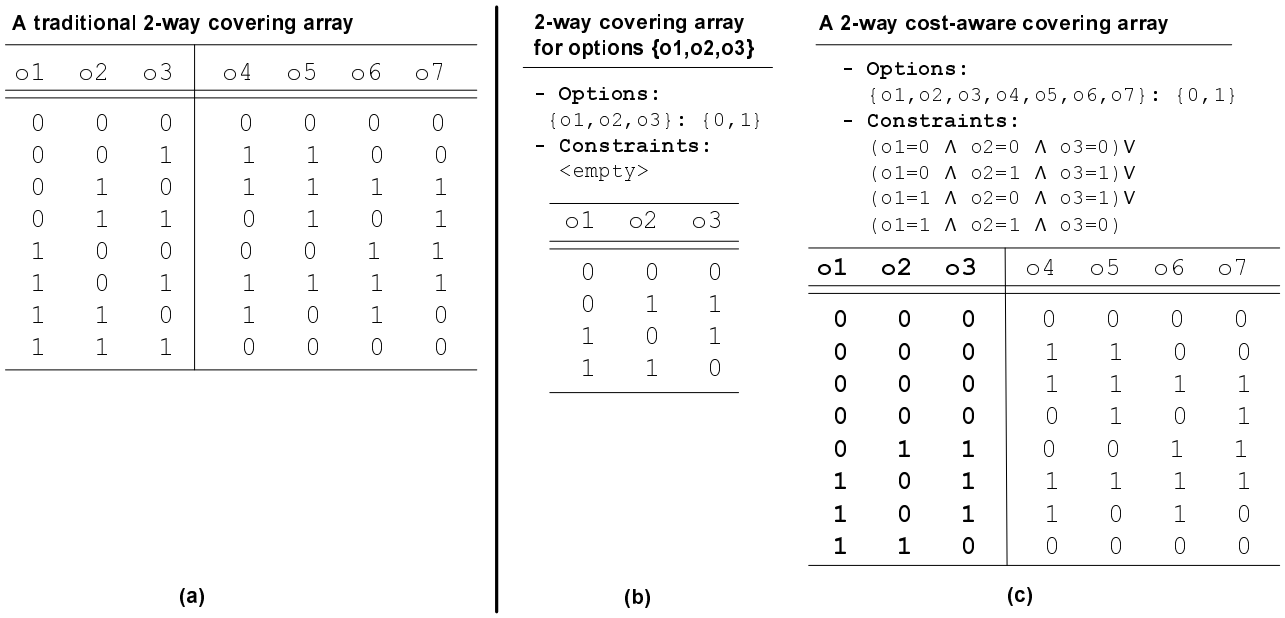
The basic justification for covering arrays is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. The results of many empirical studies strongly suggest that a majority of option-related failures in practice are caused by the interactions among only a small number of configuration options and that traditional t-way covering arrays, where t is much smaller than the number of options, are an effective and efficient way of revealing such failures [2], [6], [9], [10].

Existing approaches construct a t-way covering array in such a way that all valid t-way combinations of option settings are covered by using a minimum number of configurations. By doing so, these approaches implicitly assume a simple cost model where the cost of configuring the system under test is the same for all configurations.

In this work, we argue that this cost model is not always valid in practice. First, we observe that the configuration cost often varies from one configuration to other. For example, in a study conducted on MySQL – a widely-used and highly-configurable database management system, we observed that the cost of configuring the MySQL Community Server (a core component of the system) with its default configuration took about 6 minutes on average (on an 8-core Intel Xeon 2.53GHz CPU with 32 GB of RAM, running CentOS 6.2 operating system). On the other hand, configuring the system with NDB cluster storage support – a feature that enables clustering of in-memory databases, and with embedded server support – a feature that makes it possible to run a full-featured MySQL server inside a client application, took about 9 minutes, as these features needed to be compiled into the system. Therefore, in a covering array, reducing the number of configurations that include these features, without adversely affecting the coverage of option setting combinations, can significantly reduce the amount of time required for testing. However, existing approaches do not take actual testing costs into account when computing covering arrays.

Second, we observe that highly configurable systems often have reusable components, which, once configured, can be used in other configurations with no or very little additional cost. One simple example is the presence of compile-time

 ISBN: 978-1-61208-233-2



A traditional 2-way covering array

o1	o2	o3	o4	o5	o6	o7
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	0	1	1	1	1
0	1	1	0	1	0	1
1	0	0	0	0	1	1
1	0	1	1	1	1	1
1	1	0	1	0	1	0
1	1	1	0	0	0	0

(a)

2-way covering array for options {o1,o2,o3}

- Options:
  {o1,o2,o3}: {0,1}
- Constraints:
  <empty>

o1	o2	o3
0	0	0
0	1	1
1	0	1
1	1	0

(b)

A 2-way cost-aware covering array

- Options:
  {o1,o2,o3,o4,o5,o6,o7}: {0,1}
- Constraints:
  (o1=0 ∧ o2=0 ∧ o3=0)∨
  (o1=0 ∧ o2=1 ∧ o3=1)∨
  (o1=1 ∧ o2=0 ∧ o3=1)∨
  (o1=1 ∧ o2=1 ∧ o3=0)

o1	o2	o3	o4	o5	o6	o7
0	0	0	0	0	0	0
0	0	0	1	1	0	0
0	0	0	1	1	1	1
0	0	0	0	1	0	1
0	1	1	0	0	1	1
1	0	1	1	1	1	1
1	0	1	1	0	1	0
1	1	0	0	0	0	0

(c)

Figure 1. (a) A traditional 2-way covering array. (b,c) Illustrates our algorithm where (b) shows 2-way covering array for only compile-time options and (c) shows 2-way cost-aware covering array.

and runtime configuration options.

Compile-time options need to be set before the system can be built. The system is then configured as a part of the build process. Therefore, changing the setting of a compile-time option requires a partial or a full rebuild of the system. On the other hand, given a build of the system, runtime options are set when the system is running and the system is configured on the fly. Note that a build of the system is a reusable component. Once the system is built for a given combination of compile-time option settings, the same build can be used with different runtime configurations without any additional cost; as long as the settings of compile-time options stay the same, the same binaries can be reused. However, runtime configurations are not reusable. Even for the same build (i.e., the same compile-time configuration) they need to be reconfigured every time the program is executed, unless the program state is saved for future use.

Figure 1(a) and 1(c) illustrate the effect of reusable components on testing cost in a hypothetical scenario. In this scenario, we have 7 configuration options $o1, \ldots, o7$, each of which can take on a binary value (i.e., 0 or 1). The first 3 options $o1$, $o2$, and $o3$ are compile-time options, whereas the remaining options $o4$, $o5$, $o6$, and $o7$ are runtime options. There are no system-wide inter-option constraints; all option setting combinations are valid. Furthermore, the system is to be tested with a 2-way covering array. Two covering arrays are created for comparison.

The 2-way covering array presented in Figure 1(a) includes 8 unique combinations of compile-time option settings, requiring to build the system 8 times. On the other hand, the 2-way covering array presented in Figure 1(c) requires to build the system only 4 times, as it includes 4 unique compile-time configurations. For example, once the system is built for $o1$=0, $o2$=0, and $o3$=0, the same binaries are reused without any additional cost for 3 more configurations included in the covering array. Assuming that the runtime configuration cost is negligible compared to the compile-time configuration cost and that the compile-time configuration cost is the same for all configurations, the 2-way covering array in Figure 1(c) tests all 2-way option setting combinations at half of the cost compared to the 2-way covering array in Figure 1(a).

To improve the cost-effectiveness of CIT approaches, we introduce a novel combinatorial object, called a *cost-aware covering array*. Given a traditional configuration space model augmented with a cost function and a value of t, a t-way cost-aware covering array is a t-way covering array that minimizes the cost function. We, furthermore, provide an algorithm to compute cost-aware covering arrays for a simple, yet frequently-faced scenario in practice. The results of our empirical studies suggest that cost-aware covering arrays, depending on the configuration space model used, can greatly reduce the actual cost of testing compared to traditional covering arrays.

The remainder of the paper is organized as follows: Section II discusses related work; Section III introduces cost-aware covering arrays; Section IV presents an algorithm to compute cost-aware covering array for a particular cost model; Section V describes the empirical studies; Section VI presents concluding remarks and directions for future work.

 ISBN: 978-1-61208-233-2

II. Related Work

In this section, we provide background information on traditional covering arrays and discuss related work.

Traditional CIT approaches take as input a configuration space model $M=<O, V, Q>$. The model includes a set of configuration options $O=\{o_1, o_2, \ldots, o_n\}$, their possible values $V=\{V_1, V_2, \ldots, V_n\}$, and some system-wide inter-option constraints Q (if any). Each configuration option o_i $(1 \leq i \leq n)$ takes a value from a finite set of $|V_i|$ distinct values $V_i = \{v_{i1}, v_{i2}, \ldots, v_{i|V_i|}\}$.

Definition 1. *Given a configuration space model $M=<O, V, Q>$, a t-tuple $\phi_t=\{<o_{i_1}, v_{j_1}>, <o_{i_2}, v_{j_2}>, \ldots, <o_{i_t}, v_{j_t}>\}$ is a set of option-value tuples for a combination of t distinct options, such that $1 \leq t \leq n$, $1 \leq i_1 < i_2 < \ldots < i_t \leq n$, and $v_{j_p} \in V_{i_p}$ for $p=1, 2, \ldots, t$.*

Not all the t-tuples may be valid due to the constraints Q. Let $valid(\phi_t, Q)$ be a deterministic function such that $valid(\phi_t, Q)$ is true, if and only if, ϕ_t satisfies the constraint Q. Otherwise, $valid(\phi_t, Q)$ is false. The set of all *valid* t-tuples Φ_t under constraint Q is then defined as: $\Phi_t=\{\phi_t : valid(\phi_t, Q)\}$.

Definition 2. *Given a configuration space model $M=<O, V, Q>$, a valid configuration c is a valid n-tuple, i.e., $c \in \Phi_n$, where $n = |O|$.*

Definition 3. *Given a configuration space model $M=<O, V, Q>$, the valid configuration space C is the set of all valid configurations, i.e., $C=\{c : c \in \Phi_n\}$.*

Definition 4. *A t-way covering array $CA(t, M=<O,V,Q>)$ is a set of valid configurations, in which each valid t-tuple appears at least once, i.e., $CA(t, M=<O, V, Q>)=\{c_1, c_2, \ldots, c_N\}$, such that $\forall\, \phi_t \in \Phi_t\, \exists\, c_i \supseteq \phi_t$, where $c_i \in C$ for $i=1, 2, \ldots, N$.*

The problem of generating covering arrays is NP-hard [15]. Nie et al. classify the methods for generating covering arrays into 4 main categories [15]: random search-based methods [16], heuristic search-based methods [8], [4], [7], [11], [4], [17], greedy methods [6], [9], [5], [19], [18], [14], and mathematical methods [20], [13], [21], [12].

Random search-based methods employ a random selection without replacement strategy [16]. Valid configurations are randomly selected from the configuration space in an iterative fashion until all the required t-tuples have been covered by the configurations selected.

Heuristic search-based methods, on the other hand, employ heuristic search techniques, such as hill climbing [8], tabu search [4], and simulated annealing [7], or AI-based search techniques, such as genetic algorithms [11] and ant colony algorithms [17], to compute covering arrays. These methods typically maintain a set of configurations at any given time and iteratively apply a series of transformations to the set until the set constitutes a t-way covering array.

Greedy algorithms also operate in an iterative manner [6], [9], [5], [19], [18], [14]. At each iteration, among the sets of configurations examined as candidates, the one that covers the most previously uncovered t-tuples is included in the covering array and the newly covered t-tuples are then marked as covered. The iterations end when all the required t-tuples have been covered.

Mathematical methods for constructing covering arrays have also been studied [20], [13], [21]. Some mathematical methods are based on recursive construction methods, which build covering arrays for larger configuration space models (i.e., the ones with a larger number of configuration options) by using covering arrays built for smaller configuration space models (i.e., the ones with a smaller number of configurations) [20], [13]. Other mathematical methods leverage mathematical programming, such as integer programming, to compute covering arrays [21].

Our approach differs from existing covering array generators in that we compute a t-way covering array that minimizes a given cost function, rather than minimizing the number of configurations required.

Furthermore, Bryce et al. introduce the concept of "soft constraints" to mark option setting combinations that are permitted, but undesirable to be included in a covering array [3]. Although soft constraints could be used to avoid costly combinations of options settings, thus to reduce testing cost, using soft constraints for this purpose can be considered to be an opportunistic approach. Our approach, on the other hand, takes the task of reducing the cost as an optimization criterion.

III. Cost-Aware Covering Arrays

In our approach, we take as input a traditional configuration space model augmented with a cost function $cost(.)$. Given a covering array ca, $cost(ca)$ returns the expected cost of testing ca.

Definition 5. *Given a configuration space model $M=<O, V, Q, cost(.)>$ and a value of t, a t-way cost-aware covering array is a t-way covering array that minimizes the function $cost(.)$.*

Defining the cost function is not a trivial task. For example, the cost of a given covering array may not simply be the sum of the cost of the configurations in the array, as some parts of a configured system can be reused by other configurations with no or little additional cost. Therefore, we present a framework for defining the cost function.

Definition 6. *Given a configuration space model $M=<O, V, Q>$, a* component class *$X=\{o_{i_1}, o_{i_2}, \ldots, o_{i_k}\}$ is a set of k distinct options, such that $X \subseteq O$.*

 ISBN: 978-1-61208-233-2

Definition 7. *Given a* component class $X=\{o_{i_1}, o_{i_2}, \ldots, o_{i_k}\}$, *a* component x *is a k-tuple of the form* $\{<o_{i_1}, v_{j_1}>, <o_{i_2}, v_{j_2}>, \ldots, <o_{i_k}, v_{j_k}>\}$ *for the configuration options included in* X, *where* $k=|X|$.

We assume that the set of configuration options O are divided into p ($1 \leq p \leq |O|$) component classes $X_1, X_2, \ldots, X_p$, such that $X_i \cap X_j = \emptyset$ for $i \neq j$ and $X_1 \cup \ldots \cup X_p = O$. Consequently, a given configuration c is composed of p components $x_1, x_2, \ldots, x_p$, such that x_i is a component of component class X_i for $i=1, \ldots, p$.

For example, in our running example depicted in Figure 1, we have two component classes: $X_1=\{o1, o2, o3\}$ and $X_2=\{o4, o5, o6, o7\}$. X_1 includes all the compile-time options, whereas X_2 includes all the runtime options.

We distinguish between two types of component classes: reusable and non-reusable component classes.

Definition 8. *A reusable component class* X^r *is a component class whose components can be configured in isolation and, once configured, they can be reused in other configurations.*

Definition 9. *A non-reusable component class* X^{nr} *is a component class whose components need to be configured every time they are used.*

Going back to our running example, we observe that X_1 is a reusable component class, since, once the system is built for a given compile-time configuration, the resulting binaries can be reused in other configurations with different runtime configurations. On the other hand, X_2 is a non-reusable component class, since the runtime options need to be configured every time the system is executed.

To determine the cost of a given covering array, we assume two cost functions: $cc(.)$ and $lc(.)$. The function $cc(x)$ takes as input a component x (either a reusable or a non-reusable component) and returns the configuration cost of x. For example, assuming that the reusable component x represents a configuration for a library, $cc(x)$ is the cost of compiling the library with the given configuration. The function $lc(c)$, on the other hand, takes as input a configuration c and returns the cost of linking (i.e., gluing) together the components appearing in the configuration. For example, assuming that a configuration c is composed of reusable components x_1^r and x_2^r, each of which represents a library, $lc(c)$ is the cost of linking the two libraries after they are compiled, i.e., after the $cc(x_1^r)$ and $cc(x_2^r)$ costs are paid.

Definition 10. *The cost of a configuration* c, *which is composed of components* $x_1, x_2, \ldots, x_p$, *is defined as*

$$\left(\sum_{1 \leq i \leq p} cc(x_i)\right) + lc(c)$$

However, in the presence of reusable components, the cost of a given covering array is *not* the sum of the cost of the configurations included in the array.

Definition 11. *Given a covering array* $ca=\{c_1, c_2, \ldots, c_N\}$, *let* R_i *and* S_i *be the set of reusable and non-reusable components in a configuration* c_i, *respectively, where* $1 \leq i \leq N$. *The cost of the covering array* ca *is then defined as follows:*

$$cost(ca)=\sum_{x \in \bigcup_{1 \leq i \leq N} R_i} cc(x) + \sum_{1 \leq i \leq N} \left(lc(c_i) + \sum_{x \in S_i} cc(x)\right)$$

Furthermore, reusable components can form a hierarchy.

Definition 12. *A reusable composite component is a component, which is composed of reusable components and/or other reusable composite components.*

Reusable composite components are constructed by linking the components appearing in the composite, once these components are configured. Therefore, to account for composite components, the $lc(.)$ function should ensure that the linking cost of the same reusable composite components is paid only once.

IV. Computing Cost-Aware Covering Arrays for a Simple Cost Model

We conjecture that all the methods that have so far been used to compute traditional covering arrays, such as random search-based methods, heuristic search-based methods, greedy methods, and mathematical methods (Section II), can also be used to construct cost-aware covering arrays, all with their own pros and cons. In this work, however, we, as a proof of concept, present an algorithm to compute cost-aware covering arrays for a simple, yet important cost model.

In this cost model, the system under test has compile-time and runtime options. For a given configuration space model of the system, we define two components X^r and X^{nr}. X^r is a reusable component class, containing all the compile-time options in the model, whereas X^{nr} is a non-reusable component class, containing all the runtime options in the model. We assume that (1) the cost of linking compile-time and runtime configurations is negligible, i.e., $lc(c)=0$ for all c, (2) the compile-time configuration cost is the same for all compile-time configurations, i.e., $cc(x^r)=a$ for some constant a for all x^r, and (3) the runtime configuration cost of the system is negligible, i.e., $cc(x^{nr})=0$ for all x^{nr}.

Under this cost model, the cost of a covering array $ca=\{c_1, c_2, \ldots, c_N\}$ is

$$cost(ca) = a \times |\bigcup_{1 \leq i \leq N} R_i|, \quad (1)$$

where a is the constant cost of building the system, and R_i is the set of compile-time components appearing in configuration c_i ($1 \leq i \leq N$). In other words, under this model the optimization criterion is to minimize the number of times the system is built, while covering all t-tuples.

Although this cost model may seem to be overly constrained at a first glance, since our goal is to demonstrate the differences between the cost-effectiveness of traditional and cost-aware covering arrays, rather than to compute cost-aware covering arrays for any given cost function, we believe that the cost model employed serves well to its purpose.

Furthermore, based on our feasibility studies conducted on MySQL – a highly-configurable database management system, and Apache – a highly-configurable HTTP server, we argue that this simple cost model still has some practical importance. For example, we observed that (1) both subject applications have compile-time and runtime options, (2) runtime configuration cost for both subject applications is negligible, (3) the cost of linking runtime configurations with compile-time configurations is negligible. Although, for both subject applications, compile-time configuration costs vary from one configuration to another, since building these systems from scratch is costly, reducing the number of times they are built is still of practical value, e.g., building the entire software suite that comes with the source code distribution of our subject applications with the default configuration takes about 80 minutes for MySQL and 8 minutes for Apache, on average.

With all these in mind, Algorithm 1 presents our algorithm. In this algorithm, we use traditional covering array construction as a computational primitive. In particular, we assume a generator $\prod(t, M)$ that constructs a traditional t-way covering array for the configuration space model M.

Given a configuration space model M and a value of t, our algorithm operates as follows: (1) a traditional t-way covering array Ω is generated for only the compile-time options (line 1), (2) all the compile-time configurations included in the newly computed array are expressed as an inter-option constraint Q (line 3-5), (3) a traditional t-way covering array Ψ satisfying Q, is generated for all the configuration options (line 6). The output Ψ (line 7) is a t-way cost-aware covering array, aiming to minimize the testing cost, i.e., aiming to minimize the number of times the system is required to be built.

The rationale behind this algorithm is a simple one. Assuming $\prod(t, M)$ generates a traditional t-way covering Ω for only the compile-time options with minimal number of configurations, Step (1) selects a minimal set of compile-time configurations covering all t-way combinations of option settings for the compile-time options. Step (2), by expressing these compile-time configurations as constraints, ensures that step (3) computes a traditional t-way covering array around these configurations without introducing new compile-time configurations, minimizing the number of compile-time configurations required, thus the testing cost. If the traditional covering array generator $\prod$ produces a sub-optimal solution, then so will our algorithm.

Figure 1(b) and (c) illustrate the algorithm in our running example introduced in Section I. First, a traditional 2-way covering array is generated for the 3 compile-time options $o1$, $o2$, and $o3$ (Figure 1b). The array has 4 compile-time configurations. Second, these configurations are expressed as a constraint so that no additional compile-time configurations can be selected (Figure 1c). Finally, a traditional 2-way covering array satisfying the constraint is generated for all the options. The resulting cost-aware covering array requires to build the system under test 4 times.

Algorithm 1 Computes a t-way cost-aware covering array

Input M=<$O, V, \emptyset$>: Configuration space model
Input t: Covering array strength
Let M' be the configuration space model for only the compile-time options in M

1: $\Omega \leftarrow \prod(t, M')$
2: $Q \leftarrow \emptyset$
3: **for each** $c = \{<o_{i_1}, v_{j_1}>, <o_{i_2}, v_{j_2}>, \ldots\}$ **in** Ω **do**
4: $\quad Q \leftarrow Q \vee \{o_{i_1} = v_{j_1} \wedge o_{i_2} = v_{j_2} \wedge \ldots\}$
5: **end for**
6: $\Psi \leftarrow \prod(t,$ M=<O, V, Q>)
7: **return** Ψ

V. EXPERIMENTS

To evaluate the proposed approach, we conducted a set of experiments.

A. Experimental Setup

To carry out the experiments, we first implemented our algorithm. In the implementation, we used a well-known and widely-used covering array generator ACTS (v1.r9.3.2) [1].

We then determined a configuration space model for a hypothetical system and varied the model in a systematic and controlled manner to obtain other models. For each configuration space model obtained, we computed a traditional t-way covering array and a t-way cost-aware covering array, and compared their cost-effectiveness, i.e., compared the number of builds required by these arrays. The constant cost we assume for each configuration (build) would vary for different systems but this does not affect our cost comparisons.

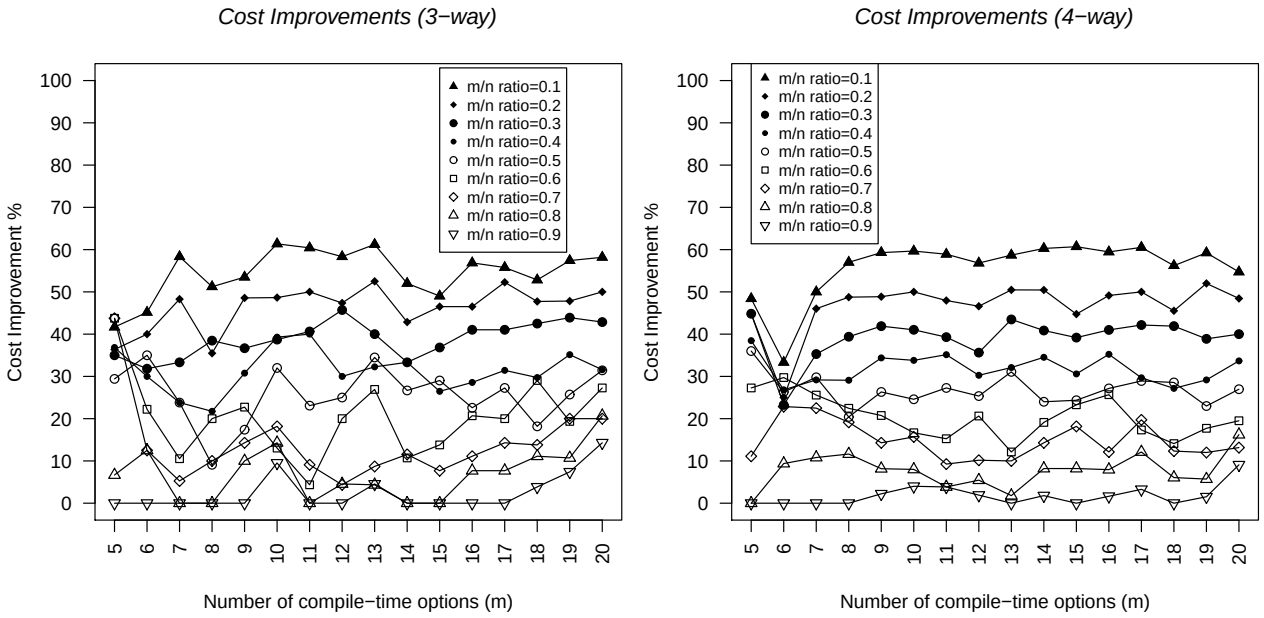
All the experiments were performed on an 8-core Intel Xeon 2.53 GHz CPU platform with 32 GB of RAM, running CentOS 6.2 operating system.

B. Independent Variables

In particular we experimented with 3 independent variables:

- m: The number of compile-time options in the configuration space model. We experimented with m=$5, 6, \ldots, 20$.

 ISBN: 978-1-61208-233-2



(a) (b)

Figure 2. Cost Improvements in a) 3-way b) 4-way cost-aware covering arrays with different m.

- m/n: The ratio of compile-time options to the total number of options in the configuration space model, where n is the total number of options and $n - m$ is the number of runtime options. We experimented with m/n=$0.1, 0.2, \ldots, 0.9$.
- t: The strength of the covering array. We experimented with t=$3, 4$.

In all the configuration space models, we, without losing the generality, used binary options only. Given m and m/n ratio, the respective configuration space model is obtained by adding binary runtime options to the model, such that the requested m/n ratio is attained. Furthermore, we opted not to experiment with t=2 because for the m and m/n values used in the experiments, the sizes of the covering arrays generated were similar to each other. This made it difficult to analyze the effect of our independent variables on the cost-effectiveness of cost-aware covering arrays.

C. Evaluation Framework

To evaluate the cost-effectiveness of cost-aware covering arrays and compare it to that of traditional covering arrays, we counted the number of unique compile-time configurations required by the arrays. That is, we counted the number of times the system is required to be built. Note that this is indeed the optimization criterion dictated by the cost model our algorithm is designed for (Section IV).

When creating the traditional covering arrays, we configured ACTS to create partially filled covering arrays. In a partially filled covering array, some option settings are left unset, indicating that, regardless of the actual settings used for these, as long as they are valid settings for the respective options, the array will still be a covering array. Once a partially filled traditional covering array was obtained, we followed a greedy approach to pick the best settings for the unset options so that the number of compile-time configurations is reduced as much as possible. Had we had ACTS to create fully filled covering arrays, the unset options would have been randomly set, which could have increased the number of compile-time configurations required. Therefore, the fully filled traditional covering arrays used in the comparisons represent the best case scenario for the partially filled covering arrays created by ACTS.

D. Data Analysis

Figure 2a-b present the results we obtained. In these figures, the horizontal axis denotes the values of m (i.e., the number of compile-time options) used in the experiments, whereas the vertical axis depicts the percentage of cost improvements (i.e., percentage of decrease in the number of compile-time configurations required) provided by cost-aware covering arrays over traditional covering arrays. Figure 2a is for t=3 and Figure 2b is for t=4.

We first observed that the cost-effectiveness of cost-aware covering arrays were better or the same (but never worse) compared to that of traditional covering arrays. More accurately, when t=3, the cost-effectiveness of cost-aware covering arrays were better than that of traditional covering arrays in 89% (128 out of 144) of the comparisons. In the

Table I
3-WAY AND 4-WAY COST IMPROVEMENT (%) AVERAGES FOR DIFFERENT M/N RATIOS.

m/n ratio	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
3-way	54.58	46.31	38.86	31.31	25.63	20.27	14.03	6.89	2.48
4-way	55.83	46.80	39.25	31.83	26.88	20.45	14.80	7.72	1.83

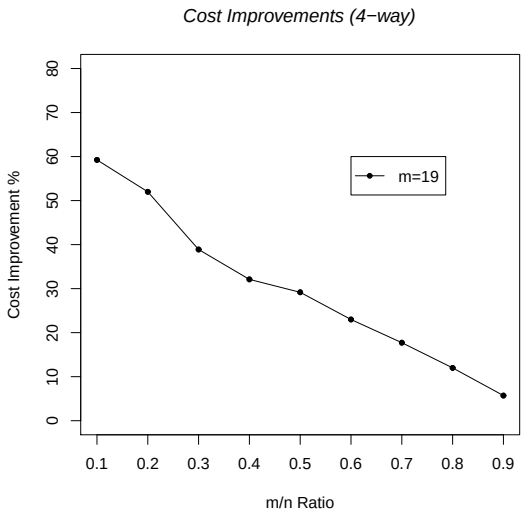


Figure 3. Cost Improvements in 4-way cost-aware covering arrays with different m/n ratio for $m = 19$.

remaining comparisons (i.e., 11% of the comparisons), the cost-effectiveness of the arrays were the same. When t=4, the cost-effectiveness of cost-aware covering arrays were better in 94% and the same in 6% of the comparisons.

We then observed that actual cost improvements varied depending on the m/n ratio used in the configuration space models. For a fixed m, as the m/n ratio increased, cost improvements tended to decrease. Table I presents the cost improvement percentages. For example, when t=4 and m=19, the cost-aware covering arrays, compared to the traditional covering arrays, reduced the cost by 59.24%, 52%, 38.89%, 32.1%, 29.17%, 22.99%, 17.72%, 12%, 5.71% when m/n=0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, respectively (Figure 3). Clearly, when m/n=1, regardless of the value of m, as the configuration space model will include only compile-time options, there will be no difference between the cost-effectiveness of traditional and cost-aware covering arrays.

For the values of m and m/n used, when the m/n ratio was fixed, the cost improvements tended to be stable regardless of the value of m. On the other hand, when $m \leq t$, as the compile-time configurations will be tested significantly, there will be no difference between the cost-effectiveness of traditional and cost-aware covering arrays.

Furthermore, comparing 4-way and 3-way cost-aware covering arrays with traditional covering arrays, we observed that 4-way cost-aware covering arrays tended to provide slightly more cost improvements than 3-way cost-aware covering arrays; as t was increased from 3 to 4, the cost improvements over traditional covering arrays tended to increase (Table I). For example, when m/n=0.1, the average cost improvement provided by 3-way cost-aware covering arrays was 54.58%, whereas 4-way cost-aware covering arrays provided 55.83% cost improvement.

VI. CONCLUSION AND FUTURE WORK

In this paper, we first introduced a novel combinatorial object, called a *cost-aware covering array*. Unlike traditional t-way covering arrays, which aim to minimize the number of configurations required to cover all valid t-tuples, t-way cost-aware covering arrays aim to cover all t-tuples in a set of configurations, which minimizes a given cost function. Given a set of configurations, the cost function computes the actual cost of testing. Furthermore, since computing the testing cost in configuration spaces is a nontrivial task, especially in the presence of reusable components, we provided a framework for defining the cost function. Finally, we presented an algorithm to compute cost-aware covering arrays for a particular cost scenario, and empirically evaluated the cost-effectiveness of cost-aware covering arrays.

All empirical studies suffer from threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial practice. One potential threat is that our algorithm was designed for a particular cost scenario. However, the cost scenario used in the paper, although simple, is of great practical importance.

Another possible threat to external validity concerns the representativeness of the configuration space models used in the experiments. Although we systematically varied the models and evaluated the cost-effectiveness of the proposed approach, i.e., a total of 288 different models were used (16 values of m × 9 values of m/n × 2 values of t), these models are still one suite of models. A related issue is that the configuration space models used in the experiments did not contain any inter-option constraints. While these issues pose no theoretical problems (our algorithm can be modified to account for constraints), we need to apply our approach to more realistic configuration space models in future work.

Despite these limitations, we believe our study supports our basic hypotheses. We reached this conclusion by noting that our studies showed that: (1) in practice, the testing cost

 ISBN: 978-1-61208-233-2

may not be the same for all configurations, (2) accounting for the presence of reusable components, i.e., the components, which, once configured, are reused in other configurations, can reduce the testing cost, (3) minimizing the number of configurations as is the case in traditional covering arrays does not necessarily minimize the actual cost of testing, and (4) the cost-aware covering arrays were generally more cost-effective than the traditional covering arrays used in the experiments.

We believe that this line of research is novel and interesting, but much work remains to be done. We are therefore continuing to develop new approaches that overcome existing limitations and threats to external validity. In particular, we are developing tools and algorithms that are based on metaheuristic search techniques, such as simulated annealing, to compute cost-aware covering arrays for any given configuration space model and for any cost function.

REFERENCES

[1] Advanced Combinatorial Testing System (ACTS), 2010. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html 10.23.2012.

[2] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[3] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960 – 970, 2006. Advances in Model-based Testing.

[4] R. C. Bryce and C. J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1082–1089, New York, NY, USA, 2007. ACM.

[5] R. C. Bryce and C. J. Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Softw. Test. Verif. Reliab.*, 19:37–53, March 2009.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–44, 1997.

[7] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 394–405, Washington, DC, USA, 2003. IEEE Computer Society.

[8] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[9] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proc. of the 24th Pacific Northwest Software Quality Conference*, pages 285–294, 2006.

[10] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the Int'l Conf. on Software Engineering*, pages 285–294, 1999.

[11] S. Ghazi and M. Ahmed. Pair-wise test coverage using genetic algorithms. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 2, pages 1420–1424, Dec. 2003.

[12] A. Hartman. Software and hardware testing using combinatorial covering suites. In M. C. Golumbic and I. B.-A. Hartman, editors, *Graph Theory, Combinatorics and Algorithms*, volume 34 of *Operations Research/Computer Science Interfaces Series*, pages 237–266. Springer US, 2005.

[13] N. Kobayashi. *Design and evaluation of automatic test generation strategies for functional testing of software*. Osaka University, Osaka, Japan, 2002.

[14] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog-ipog-d: efficient test generation for multi-way combinatorial testing. *Softw. Test. Verif. Reliab.*, 18:125–148, September 2008.

[15] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43:11:1–11:29, February 2011.

[16] P. J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 49–59, Washington, DC, USA, 2004. IEEE Computer Society.

[17] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '04, pages 72–77, Washington, DC, USA, 2004. IEEE Computer Society.

[18] K.-C. Tai and Y. Lei. A test generation strategy for pairwise testing. *Software Engineering, IEEE Transactions on*, 28(1):109 –111, Jan 2002.

[19] Y.-W. Tung and W. Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431 – 437 vol.1, 2000.

[20] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques*, TestCom '00, pages 59–74, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[21] A. W. Williams and R. L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, TestCom '02, pages 283–298, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

 ISBN: 978-1-61208-233-2

Sick But Not Dead Testing -
A New Approach to System Test

Tara Astigarraga[1], Michael Browne[2], Lou Dickens[3],
Systems and Technology Group
IBM
[1] Rochester, NY 14626
[2] Poughkeepsie, NY 12601
[3] Tucson, AZ 85744
{asti, browne, dickens}@us.ibm.com

Abstract— **Enterprise data center implementations make significant investments in high availability configurations, redundant hardware, software and Input / Output (I/O) paths that are in many failure scenarios quite successful. However, in spite of all that investment clients are still facing unexpected outages and performance impacts related to a phenomenon referred to as Sick but not Dead (SBND) errors. SBND errors are sometimes lumped together in a category with other related errors including transient errors, partial failure scenarios and soft errors. While SBND errors do have many common characteristics with the errors described above, there are key differences and environment impacts which we will explore further in this paper. We will also present new proactive techniques, inject scenarios and methods to identify, characterize and address SBND failures including cross-component impacts and failures.**

Keywords-Software Testing; Sick but not Dead; Software Engineering; Partial Failure; Transient Error; Soft Failure; SAN Test; System Test.

I. INTRODUCTION AND MOTIVATION

Despite high availability (HA) configurations, customers are still experiencing outages and severe performance declines in their environments. These outages typically show no signs of hard component failures for which the HA infrastructure would react to and provide recovery. We classify these errors as Sick but not Dead (SBND) failures. These errors are often the hardest failures to identify and can have sporadic but lasting impacts on the environment as a whole. SBND failures currently represent 80% of business impact, but only about 20% of the problems [2].

SBND errors are sometimes lumped together in a category with other related errors including transient errors, partial failure scenarios and soft errors. While SBND errors do have many common characteristics with the errors described above, there are key differences as well. SBND errors by definition derive from a component within the I/O path that is 'sick' meaning behaving in an unorthodox or partially failed fashion but not completely 'dead' or hard failed. Depending on the component exhibiting the SBND characteristics, the symptoms can vary, come and go at different intervals and it can take anywhere from seconds to months for the component to finally reach a hard fail state. It is this in-between time when the component is defined as SBND.

Complex customer solutions and environments utilizing mixed vendor products and technologies create textbook scenarios for SBND failures to occur. Many products are intolerant of misbehavior of other devices and most failure paths deal promptly with hard failure scenarios, but are slower and more cautious to react to partially failed, misbehaving, or SBND components in a Storage Area Network (SAN). With current field solutions, problem determination related to SBND failure scenarios is complex, time consuming and often requires special problem determination lab trace tools and a team of cross-vendor product and solution experts. Current resolutions to SBND failure scenarios are almost always reactive vs. proactive. In our system test and SAN labs we have been developing new proactive techniques, protocol inject scenarios and methods to identify, characterize and address SBND failures including cross-component impacts and failures across the I/O path.

Our current research related to SBND defects reported shows that the highest number of SBND problems exists along the I/O path. While related problems do occasionally exist within specific internal sever paths they are significantly less frequent, easier to debug and typically contained to a single server and handled via embedded HA mechanism.

Systems generally behave properly when failures are solid or hard failures. It is when components act SBND that system availability is often at risk. In these scenarios failover or recovery mechanisms often do not behave as we should expect them to. Often times the problems are corner cases where they are not easily reproducible and hard to trouble shoot, but continue to plague customer environments. It should also be noted that SBND problems are not something that occur in a particular vendor or product set, but rather a system level event that occurs when one (or more) component(s) in the environment does not always behave consistently. Since the problem does not relate to a particular vendor or component issue it is not a simple fix but rather a system level event that must be fully understood, tested and addressed by all vendors in a distributed systems SAN environment.

The focus of this paper will be on SBND failures related to the I/O path in distributed systems Fibre Channel (FC) SAN and Fibre Channel over Ethernet (FCoE) environments. In

 ISBN: 978-1-61208-233-2

this paper we will better define and characterize SBND failures, explain the impacts they can have on complex customer environments and introduce new testing techniques and injections we have deployed in our system test labs.

II. COMMON CHARACTERISTICS OF SBND FAILURES

Most SBND failures are not obvious product failures. Often when problem determination begins all individual products in the environment may appear 'healthy' and existing internal diagnostics often do not flag anything. Even error log reviews may come up relatively clean, making problem determination very difficult. SBND problems by definition are transient errors, meaning a product is temporarily misbehaving, making the side-effects or symptoms in an environment often appear and disappear.

SBND failures are frequently first noticed at the host or application layer. The tables below outline the most frequent symptoms and characteristics displayed when SBND failures were encountered.

TABLE I. MOST FREQUENT SBND SYMPTOMS

Severe performance degradation at sporadic intervals
Mirror or replication times exceeding Service Level Agreements
I/O redrives
I/O near redrives
Application sensitivity to Recoverable I/O Events
Product interaction behaviors related to unforeseen external trigger events

TABLE II. COMMON SBND CHARACTERISTICS

Not an obvious product failure, individual products in the environment appear 'healthy' even after detailed internal dump analysis at highest levels of product support
HA Mechanisms see no error and don't react
Hard for software and monitoring products to detect, internal diagnostics often do not find anything
Problems often appear and disappear
Start slowly and often amplify with time

Note; the 2 tables above were compiled using defect data from problems that were encountered in the IBM system test labs and the IBM field support group from 2010 through 2012.

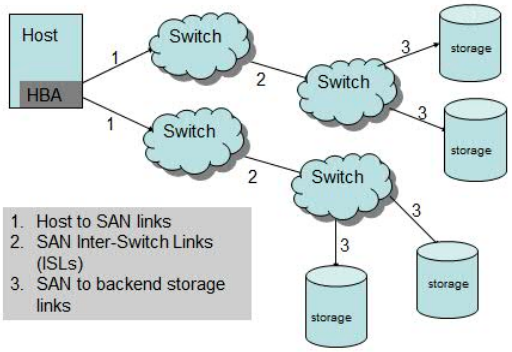
One might fail to realize the size and/or scope of a SBND failure, by examining the symptoms alone. This is because SBND failures commonly create a sympathy sickness throughout the entire network. Sympathy sickness is when a single device or condition in one part of a network impairs the performance of other devices or other parts of the network. For example, a single bad small form-factor pluggable (SFP) in one of the E-ports of an inter-switch link (ISL) can intermittently corrupt frames that are being transported through the ISL [3]. The other switches in the SAN or the end devices will discard these corrupted frames. This will result in the initiators having to perform error recover, and re-drive the corrupted I/O exchanges. Thus one bad SFP in an E-port, can affect the performance of 100's or 1000's of initiators that have their frames transported over the ISL.

III. TEST APPROACH

In a proactive attempt to better address and improve test and design around SBND customer failures, IBM introduced an internal quality improvement effort to better define, categorize and test SBND failures. As part of this ongoing effort, the IBM Systems and Technology Group labs have begun introducing a variety of SBND symptoms into complex system test environments using a three-pronged approach. 1. Build a center of competency around identifying, characterizing and debugging SBND failures in the I/O path. 2. Target modified reliability, availability and serviceability (RAS) microcode to better identify and flag SBND failures for troubled areas. 3. Targeted test case coverage related to SBND failures, symptoms and characteristics.

For this paper we will cover the 3rd prong described above related to increased SBND testing and early results. In late 2010 our SAN test labs within IBM began technical analysis on SBND errors and targeted ways to not only inject SBND failures, but to proactively monitor the environment as a whole for related defects and outages. This was a detailed and controlled approach consisting of injects in three primary locations within the I/O path, as outlined in figure 1 below.

Figure 1. Example of Typical SAN



Once the inject areas were established and test tools in place we began targeted testing covering the most frequent SBND symptoms and characteristics described in tables 1 and 2. Table 3 below outlines some of the test injects symptoms and test case examples that were created to inject SBND symptoms into our SAN environments to monitor for proper handling and unintended side effects.

TABLE III. SBND TEST SCENARIO INJECTS

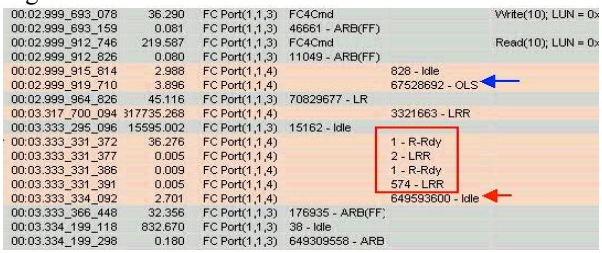
Symptom:	*Types of Injects Used:*	*Test Case Examples: [4]*
Severe Performance	1. Credit starvation 2. Inject Delay	1. Replace R_Rdy primitives with IDLE/ARB (FC), inject

Degradation		PFCs for Class 3 traffic (FCoE). 2. Hold all frames for *x* microseconds
Mirror or Replication times exceed Service Level Agreement	1. port flaps 2. drop frames 3. jitter	1. Port shut/no shut activity (FC,VFC,Eth) 2. Drop every *xth* frame in each direction 3. Corrupt sof, eof, crc and other header data
I/O redrives or near redrives	1. drop, corrupt or re-order data frames 2. short holds of frames	1. Target data frames and drop or re-order 2. Hold all data frames and/or transfer ready frames for *x* seconds.
Application sensitivity to Recoverable I/O Events	1. virtual link jams 2. link resets 3. corrupt frames	1. FDISC drops, VFC jitter, VSAN jams 2. Inject NOS, OLS, LR and/or LRR onto link 3. Corrupt bits in the FC or FCoE header and recalculate CRC
Product behaviors related to unforeseen external trigger events	1. protocol violations 2. unexpected data returns 3. partial recovery scenarios	1. Inject protocol deviations from standard and monitor destination handling 2. Return Check Cond to write exchange 3. Drop data frame, then drop subsequent ABTS, allow re-driven ABTS to flow through un-jammed.

IV. EARLY RESULTS

Overall we established a test suite consisting of over 100 unique SBND test cases, which are run in a controlled SAN environment allowing us the capabilities to inject a single error (or combination of errors) and monitor the environment as a whole. The majority of the problems we have identified are defects that would have been near impossible to detect and correlate in a customer environment. The ability to understand which variables are being injected at which time and location in the SAN and watching all associated host, switch and storage logs provides the ability to correlate and connect events that otherwise would have appeared to be non-related. Further, having packet level traces at each point in the SAN allows the ability to deep-dive into the traces. Figure 2 below illustrates one SBND inject example where every 5 min the Not_Operational primitive sequence (NOS) was injected to simulate a bouncing or partially failed port in the SAN. Figure 2 below shows the subsequent behaviors following one of the NOS injects which resulted in failed link initialization. For link initialization to complete successfully following our NOS injects the primitive sequences OLS/LR/LRR/IDLE/IDLE have to be traded sequentially. In figure 2 you can see one SAN vendor sent extra R_RDY primitives and LRRs prior to sending the final IDLE packets required to complete link initialization.

Figure 2: Protocol trace review



The protocol trace analysis results and frame level debug capabilities, provide enhanced problem determination capabilities and when combined with associated host, switch and storage logs and traces present a clear picture of the problem and greatly aid in cross-vendor problem determination.

Typical product system test environments and test plans will analyze recovery capabilities in a product or system offering along with potential implementation architectures and then inject hard errors to determine if products under test were behaving according to specification and customer requirements. A high level example would be a system test environment that had been designed and implemented with full redundancy of all components in order to minimize Service Level Agreement (SLA) violations [1]. The test engineer would then introduce failures of the components at injectable points in the configuration to validate and verify the system offering would meet SLA requirements. What this technique misses is the "almost errors" that are not specified or articulated as customer requirements. Additionally, there is some level of subjectivity to a SBND event actually occurring and convincing the designers that such a situation would exist in the real world. A test engineer also has to use reasonable judgment in designing the injection as any SBND injection can be pushed to unrealistic limits and then the test can be declared invalid. For example, when testing credit starvation one must be cautious in the rate of R_Rdy (frame buffer credit) drops that are injected as too many will cause link resets, replenishing credits back to the agreed upon limit during login. For SBND scenarios, the tester would want to identify the buffer credits allotted during login and drop R_Rdys at a rate which slowly impacts the environment without causing an immediate link reset. It is this careful balance that must be pursued in the test design and execution. Having a test engineering center of competency for SBND problems that can provide real world patterns of these injections is critical to wining the subjective discussions between test engineers and designers.

Since starting this work in 2010 we have seen a dramatic spike in internally found SBND related defects being identified and fixed in system test. In 2010 when we started this testing only 5% of the defects found in SAN system test were related to SBND error handling. In May of 2012, these defects accounted for 48% of the overall defects opened by the SAN system test teams. The defects opened are spread across multiple vendors and I/O path components including operating systems, host HBA/CNA firmware and drivers, multipath

 ISBN: 978-1-61208-233-2

drivers, SAN and FCoE switch code and storage firmware and drivers.

V. CONCLUSION AND FURTHER DEVELOPMENT

As complexity, virtualization and mixed vendor solutions continue to grow in the IT industry and customer solutions, the need for highly-skilled SBND low-level testing will also continue to increase. In an industry where quality is expected and customer defects can cause costly outages it is no longer sufficient to test products for correct recovery in hard failure scenarios. We need to continue to put increased focus on solution testing, and further on solution injects and handling of hard failures and SBND failures on any component within the environment.

As we continue to implement deeper SBND testing described in this paper, we are pursuing plans to continue this effort with a second phase targeting new inject methods and focus on spreading these testing capabilities and awareness across IBM test labs worldwide. Given the economic costs of the tools to inject SBND scenarios and the skill required we are also innovating in economically scalable methods to do this type of testing in more diverse testing and test skill environments. We also continue to drive a close-loop feedback process between IBM test, development and support teams and across OEM partners, ensuring that the SBND defects that have been found are fixed and lessons learned are applied to future product development and monitoring capabilities.

It is our hope and vision that impacts of SBND failures be understood across the industry and that more SBND testing and proactive measures are taken to help minimize the impacts these failures have on the environments of the future.

VI. ACKNOWLEDGMENTS

The authors would like to thank their employer, International Business Machines (IBM) for supporting their efforts to produce educational content. We would also like to thank those parties who provided quotations for use in this paper.

REFERENCES

[1] A. Hanemann, D. Schmitz, and M. Sailer, "A framework for failure impact analysis and recovery with respect to service level agreements," *Services Computing, 2005 IEEE International Conference on* , vol.2, no., pp. 49- 56 vol.2, 11-15 July 2005 doi: 10.1109/SCC.2005.10
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1524423&isnumber=32587 [retrieved: July, 2012]

[2] B. Rogers, "z/OS 1.11 Sysprog Goody Bag" Presented at SHARE Session 2228," March 2010. [Online]. Available: http://mobile.share.org/client_files/SHARE_in__Seattle/S2228RR092920.pdf [retrieved: July, 2012].

[3] FC-MI, ANSI Standard 3.2.14-3.2.34, 2001.

[4] FC-FS-3, ANSI Standard 5.2.4-5.2.5, 2008.

 ISBN: 978-1-61208-233-2

Test Driven Life Cycle Management for Internet of Things based Services: a Semantic Approach

Eike S. Reetz, Daniel Kümper and Ralf Tönjes
Faculty of Engineering and Computer Sciences
University of Applied Sciences Osnabrück
Osnabrück, Germany
Email: {e.reetz, d.kuemper, r.toenjes}@hs-osnabrueck.de

Anders Lehmann
Institute for Business and Technology
Århus University
Århus, Denmark
Email: anders@hih.au.dk

***Abstract*—Concepts for Internet of Things (IoT) are currently limited to particular domains and are tailored to meet only limited requirements of their narrow applications. To overcome current silo architectures, we propose a business oriented service composition of IoT enabled services with (semi-) automated model based testing capabilities. Explicit description of services as well as the target environment allows for automated design and execution of tests, hence enabling fast and robust IoT based service provision. This work proposes a semantic description of the test design and execution process to enable reasoning of test behaviour and suitability in the different phases of a service life cycle. The proposed work describes a test model and an appropriate test architecture. A first testbed implementation demonstrates their applicability. The proposed approach enriches current views of IoT architectures with knowledge from the field of service oriented architectures and makes them usable in distributed environments with partial unreliable resources by introducing a formalised integration of automated testing into the life cycle management.**

***Keywords*-model based testing; Internet of Things; life cycle management; semantic test description.**

I. Introduction

Seamless and transparent integration of smart objects into the environment is an open research topic for almost 20 years. Several aspects of a smart interaction between the virtual and the physical world from low level resource constraint sensors and actuators to high level description and interaction capabilities based on context-awareness have been proposed so far, resulting in several isolated solutions for the Internet of Things (IoT). Nevertheless, most of the proposed approaches address only the open issues of a specific application domain. Moreover, the limited interoperability between different silo solutions tends to prevent mass market services and service composition. To overcome these technological limitations, we propose a flexible service creation environment for IoT in order to dynamically design and integrate new types of services and therefore enable new business opportunities.

Due to interactions with the real world and a large variety of involved technologies, these services have to be very flexible and robust. This requires enhanced testing capabilities already included in the service creation process. Our approach is not only to pursue a test-driven service life-cycle management, but also to automate the testing process appropriately. Testing in the IoT domain is a challenging task: The diversity and distribution of involved components, as well as their unreliability, raise current research issues. Furthermore, the need for realistic conditions results in a complex testing environment. Moreover, the dynamics of the IoT environment make the development and maintenance of services an error prone challenge.

The presented concepts have been conducted in the scope of the European research project "IoT.est" (the overall project concept can be found at [1]). This paper focuses on the aspects related to automated testing by describing the current status of the approaches and their limitations.

The overall contribution of this paper can be summarises as follows: we propose a test-driven life cycle management which can be utilised for rapid IoT based service creation and deployment based on automated testing. Therefore, our overall concepts of a test-driven life cycle management, semantic descriptions of service and tests, and the derived test architecture are envisaged. We are following a Service Oriented Architecture (SOA) and, therefore, we are extending classical approaches by adding enhanced testing capabilities (e.g, test automation, emulation of network, resources and real world context), which we believe enable the applicability of SOA to the IoT domain.

The rest of the paper is structured as follows: after giving a brief overview of the State of the Art in Section II, the IoT service concept will be briefly explained in Section III. Afterwards the model based testing approach is introduced in Section IV and the test architecture is described in Section V. First prototype implementations and testing principles are then presented in Section VI. Conclusions and future work finally conclude this paper.

II. Related Work and Open Research Issues

Current solutions do not consider test-friendly automated development of services. Some approaches like the UML 2.0 Testing Profile [2] provide concepts for designing and

 ISBN: 978-1-61208-233-2

developing black-box tests, but do not provide guidance how to utilise it. The abstract UML 2.0 Testing Profile (U2TP) notation has to be transformed into a test specific programming language like TTCN-3 [3] or JUNIT [4]. TTCN-3 specifies tests and how they have to be executed. Several tools provide an environment for test creation and execution. Nevertheless, a drawback of TTCN-3 is the lack of simplicity in the generation of tests since the users need to learn a new language. Our approach tries to overcome this drawback by automated test case design and execution during service development.

Initial concepts of software engineering processes applied phase-models like the waterfall model and assume software can be split into sequential phases. Iterative models, such as the spiral model or V-model, change this paradigm by considering more iterative approaches. The V-model introduces testing to the phases of the waterfall model. In recent years, agile development processes, such as Extreme Programming, have been introduced. One important outcome of of the agile development approach is the test-first method, which aims at a test-driven target orientated service development process [5]. A Pattern Oriented Software Development (POSE) approach for web service development has been proposed by Chengjun [6]. A pattern represents components and relationships amongst them. For each identified POSE process a pattern is built; it comprises the activities, goals, architecture definition and validation. Adapting these software development processes to IoT service development rises some challenges: the support for services at different levels of granularity, the support for service composition, consistency checking, the identification of dependencies and the service development as contentious process. For automation purposes, detailed descriptions of processes and resources are crucial to control the IoT resource effects in the service life cycle, especially during the testing phase.

III. IoT Service Concept

The investigated approach of IoT enabled services shows similarities to classical service oriented architectures [7]. Our model for IoT enabled business services extends the classical consumer/provider role concept by connecting IoT resources (Inspired by the EU Project Sensei [8]) to the service component and differentiate between Atomic Service (AS) and Composite Service (CS). The AS is the smallest separable, which could be either a classical web service or an IoT service. IoT services access sensors and actuators by abstracting their interfaces and capabilities. They provide an interface based on SOA interfaces (e.g., RESTful), which enables reusability by other entities. A CS is a conjunction of atomic or composed services and integrates the business flow perspective into the service. The CSs are modelled with a Business Description Language like Business Process Model and Notation (BPMN) or Business Process Execution Language (BPEL), thus providing an abstract way to design the service and accelerating evaluation of services and integration of business logic.

IV. Model based Testing

Due to dynamic and unreliable components involved in IoT based services, efficient planning of resources for testing is required. In order to address test levels with a certain coverage, it is necessary to simplify and automate the test design and execution process. Therefore, model based testing is a promising approach to improve the IoT based services by generating test cases from models extracted from the service description. Figure 1 highlights the well known concept of model based testing. Abstract test cases can be derived from a service model, which is a partial description of the service. The figure shows an extension to this classical approach by introducing an explicit model of the environment in which the service is executed. The explicit description of an environmental model can be utilised to build components for emulating the environment in an abstract and executable way. Modelling the environment appears crucial for convincing test results due to the distribution and unreliability of IoT components as well as due to the interaction with the real world. Different to the classical SOA domain of web services, IoT services require a novel paradigm to model the expected interaction with physical and virtual objects. Enhanced models of the environment assure more sophisticated testing capabilities of resources, network and real world effects to the System Under Test (SUT).

The next subsection explains how model based testing is integrated into the life cycle management and highlights the different scopes of the testing process within various phases of the life cycle. Afterwards, a semantic test model is proposed, enabling reasoning of test behaviour and suitability in the different phases of a service life cycle with self-explanatory derivation and execution of test cases.

A. Test driven life cycle management

In order to understand why life cycle management is important, we need to keep the paradigm shift of SOA in mind. A SOA enables an improved alignment of business and IT needs. Due to the composition of services, logic can be separated from the implementation. Classical approaches tend to decide how to achieve quality of service, security, or combinations of functions at design time and thus reduce the flexibility of the business processes. Another improvement is the reduction of the time to market since the decomposition of applications into services increases sharing and reusability of services. Different stakeholders, as well as the integration of IoT enabled business services, require coordination and collaboration in terms of service design, execution and testing. This results in the need of a common understanding of the service life cycle process. In addition to these classical outcomes of a well defined life cycle, a management process enables to (semi-) automatise the test process based on the

 ISBN: 978-1-61208-233-2

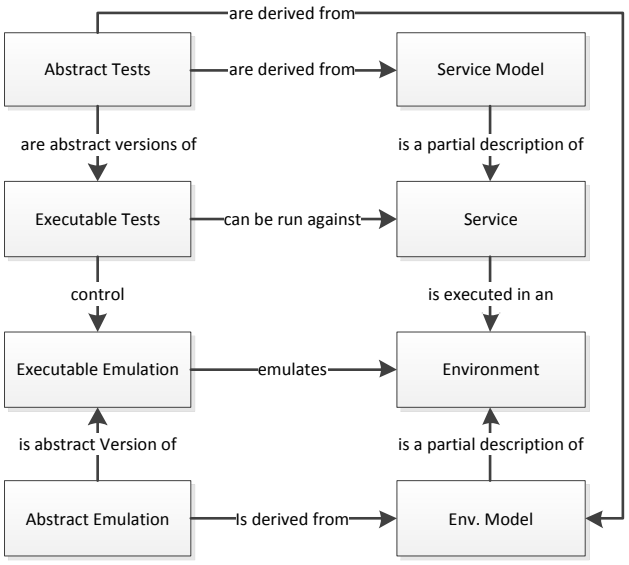


Figure 1. Extended model based testing approach

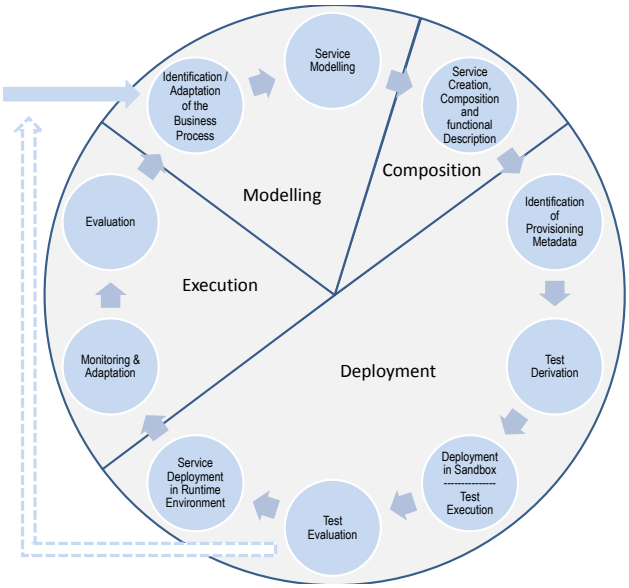


Figure 2. Test driven Life Cycle Management

semantically described process and the service itself. Therefore, knowledge about the functional and non-functional behaviour of the service as well as the test modelling and execution are explicitly described by utilising a machine and human interpretable semantic description. The proposed life cycle approach takes advantage of well known phase models. Nevertheless, we believe that the explicit integration can significantly enhance the applicability for IoT based services. This test description enables reasoning of test behaviour and suitability in the different phases of an IoT service life cycle. In these life cycle phases (Figure 2), the focus of interests is different.

In the *modelling* phase, the focus is on making the service perform according to the functional specifications. Thus, the focus is on functional testing, i.e. unit tests and integration tests.

In the *composition* phase, the focus is on building complex services by composing other services. In this phase, it becomes more important to discover the atomic services needed to achieve the goals of the composite service. Likewise, it is important to be able to discover the tests that effectively represent the composite service. These composite service tests need to make sure that the underlying services are available or can be emulated adequately. Therefore, there is a need to gather this specific service composition information and add it to the description of the tests.

In the *deployment* phase, a number of services is typically deployed. In order to be able to evaluate the success of the deployment, the semantic description of the services to be deployed can be checked for inconsistencies, contradictions and overlaps. The focus of the deployment is to prove that the deployed services will be able to deliver the services as promised in the Service Level Agreement (SLA). In the deployment phase, the concrete environment for running the service is chosen. By adding information of the concrete environment, the expected load can be determined by load tests.

In the *execution* phase, the service provider measures relevant parameters of the services being executed in order to prove compliance with the SLA. By using the information added to the load tests in the deployment phase, the service provider can predict when the services are close to reach the maximum capacity. These results can also be used to set up alarms, which will be triggered if the measured parameters indicates imminent breach of the SLA. The effect of the alarm can then lead to a dynamic re-selection of the atomic services in utilisation.

The detailed steps of the test driven life cycle management are shown in Figure 2. Its original purpose is to identify the business process requirements and goals and categorise them into different life cycle phases (short term and long term requirements). The categorisation assures a fast ability to demonstrate first results and helps to adjust requirements during the life cycle process. The next step, *Service Modelling*, decomposes the business process and tries to identify possible service components, taken into account that already available services should be reused if possible. As in all steps, requirements from previous steps are evaluated (in terms of feasibility) and new requirements are identified. Modelling goals ensures the proper identification of the required service components.

Contrary to the first steps requiring rather manual actions, the *Service Creation and Composition* phase is supported with tools to discover and compose services. The outcome of this step is a deployable service including a semantic

 ISBN: 978-1-61208-233-2

service description. The next phase takes care of required meta data for service provisioning. This includes semantic descriptions of the service contract and the service run-time where the service is be to deployed. With this information, the *Test Deviation* phase can reason about the semantic descriptions in order to build test cases as well as the test execution flow. Afterwards, the service is executed within a sandbox environment. The sandbox is controlled from the test execution engine (to be discussed in Section V) and intends to act as the service run-time environment under realistic conditions in terms of load, traffic and competing services running in parallel. The execution of test cases results in the test evaluation. The test outcome is finally compared to the expected outcome and if the tests pass successfully the service is available for deployment.

The *Service Monitoring and Adaptation* phase takes place if the service is deployed successfully. Service monitoring based on current behaviour can result in dynamic re-selection of utilised atomic services. In addition, the monitoring phase discovers if the service consumption is as expected, for example if the number of request per minute fit to the expectations. From the discoverer information further needs for the next life cycle as well as adjustments for the sandbox are detected within the *Evaluation* phase.

Contrary to classical life cycle approaches, this life cycle is driven by the semantic description of the service, the service run-time environment and the test environment. Therefore, it is possible to automatise and integrate the test design and execution into the service design-time. As a benefit there is a clear separation between developing and testing, i.e. the developer does not create the test cases explicitly, which might result in a non-optimal test coverage. Moreover, there is a kind of integrated testing since the test cases are automatically built and executed in a controllable sandbox. This results in fast feedback and rapid service improvement during design time.

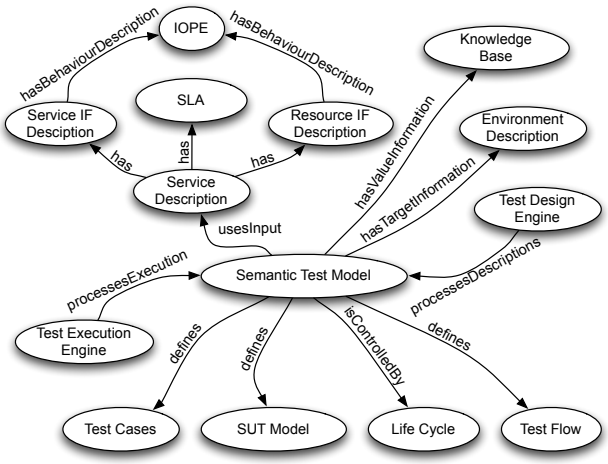


Figure 3. Semantic Test Model

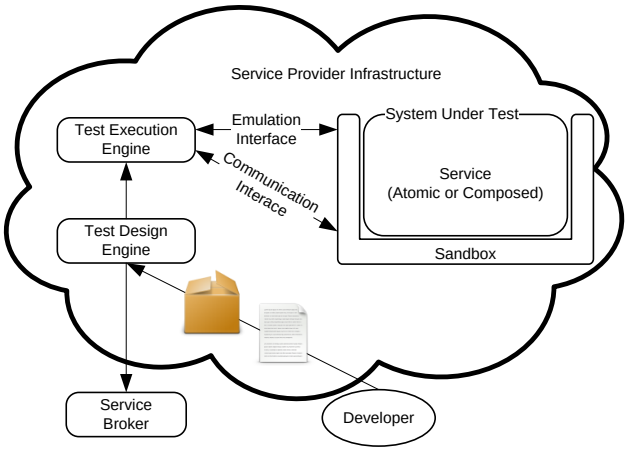


Figure 4. Test Environment for IoT enabled Composite Service

B. Semantic Test Model

The procedure for testing is closely related to service process modelling; the service models described in [9] are used to describe test cases, test data and the test flow. This work employs the concepts defined in the OWL-S [10] to specify the service test semantics – including inputs, outputs, preconditions and effects (named IOPE) used for behavioural description of the service interface and the resource interface connecting to various IoT resources.

The description of the test model within a test ontology is the basis for automated test case creation and execution. The test ontology enables the creation of reusable test cases for an SUT and the modelling of the test flow which will be executed (Figure 3). A Test Design Engine (TDE) utilises precise service descriptions and a knowledge base containing business expert knowledge, test data generation and a test oracle for deriving tests, distinguishing different types, e.g., functional or reliability, and levels of tests, i.e., unit, interface, integration, or collaboration test. The service description thereby also supports the interface description of external IoT resources used by the generic emulation interface (Section V-C). Attribute values of service descriptions are constrained by a min and max value or a value list and an optional default value. Moreover, events are defined to describe transition of states and events. Reasoning engines, e.g., rule based systems, can exploit the knowledge to derive the behaviour model and constrain the test cases, i.e., behaviour model plus test data. The defined test flow enables the Test Execution Engine (TEE) to process different tests and ensures the desired coverage regarding different states and paths of the finite state machine of the service.

The semantic test model is involved in different stages of the services and enables a highly automated test management.

 ISBN: 978-1-61208-233-2

V. TEST ARCHITECTURE

As mentioned in the previous section, due to the real world interaction and the lack of control of components involved in atomic and composite services, tests can not be executed in a productive environment. Our approach integrates systematic testing into the life cycle management. Therefore, each service that is designed will be tested in a (semi-) automated way before being deployed. The SUT will be placed in a so called sandbox, which emulates the target environment as realistically as possible – not only functionally, but also in from a real world, e.g., network and resource oriented, point of view. In order to achieve automated test case creation and execution each SUT needs to be described semantically. Although tests based on the semantic description can only detect whether the service acts as described and not as it was imagined by the developer, the test automation promises to overcome current limitations as far as complex and distributed IoT enabled composite services are concerned and can improve the service quality significantly. Figure 4 depicts the main components of the test architecture for IoT enabled services. The SUT can be either a AS or a CS. The *sandbox* ensures that the behaviour can be emulated according to the test cases. This includes the emulation of network, hardware resources and IoT resource related parameters and characteristics. The Test Execution Engine (TEE) controls the environment and executes the test cases. The TDE is responsible for deriving the test cases from the semantic description of the SUT and generates test data. The test creation process is triggered either by the upload of a new or updated service from the Service Developer (including semantic description) or by the detection of new or changed service elements in the Service Broker lookup, which might be selected from a CS at run-time. The proposed test environment is located at the service provider infrastructure of the SUT and does not consider white-box unit tests from the service developer perspective.

A. Test Design Engine

The Test Design Engine (TDE) is responsible for create test cases for new and changed services and takes care of preparing their execution. The main functions and interfaces are shown in Figure 5). The TDE is trigged via the Service Developer Design Interface by transmitting the SUT together with a semantic description. In the first phase a *Codec Plug-in Creator* will identify which protocols are utilised by the service interfaces and, if required, create a new codec in order to abstract the protocol flow. Afterwards, the test cases for functional and non-functional tests are created and the order of execution is defined based on a extended finite state machine of the service. The test cases are described with the standardised Test Control Notation Version 3 (TTCN-3) language. Afterwards, the test cases will be enriched with generated test data based on the IOPE conditions of the semantic service description. The Test Case Compiler

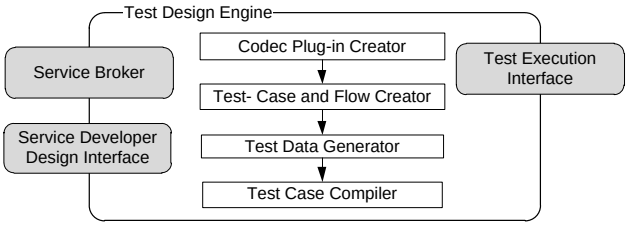


Figure 5. Test Design Engine for IoT based Composite Service

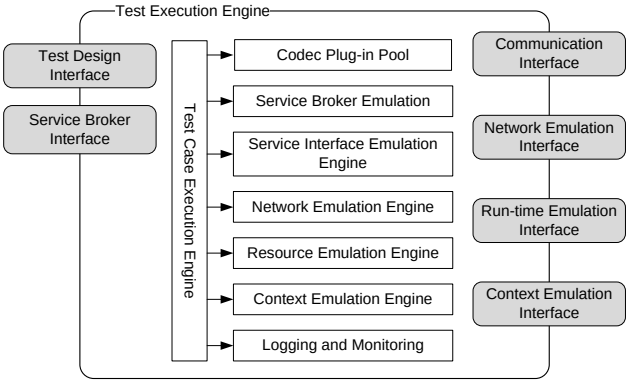


Figure 6. Test Execution Engine for IoT based Composite Service

produces executable code from the test cases and assures a native test case execution by sharing test case executable code via the Test Execution Interface with the TEE.

B. Test Execution Engine

The Test Execution Engine (TEE) is the central component to coordinate the test flow. Figure 6 depicts the main components and interfaces of it. The execution is triggered by the Test Design Interface. While the *Test Case Execution Engine* takes care of the test execution it accesses emulation components in order to execute the SUT under controlled and emulated conditions of the target environment. This includes also the emulation of the Service Broker (in case of composite services) in order to control the run-time service selection. In case of a service request from the SUT, the emulated Service Broker answers the request with the binding address of the *Service Interface Emulation Engine*. This assures that no external resources are involved in the execution and allows for testing all possibly correct and partly incorrect behaviours of the external service with fully controllable emulated components. Therefore, the test execution of composite service is capable of testing the interoperability of the connected services without directly executing the involved components.

C. Sandbox

The sandbox ensures that the SUT can be executed in a test environment and can be manipulated during the test execution (shown in Figure 7). In addition, the separation between the TEE, and the sandbox offers the ability to execute

 ISBN: 978-1-61208-233-2

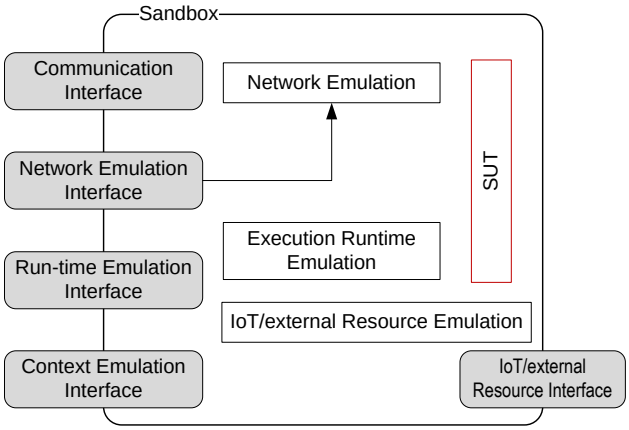


Figure 7. System Under Test Encapsulation

the tests in a distributed manner. The SUT service interfaces are connected with the Test Encapsulation Communication interface via the *Network Emulator*. Each message from or to the SUT can be manipulated in terms of delay and packet loss for evaluating the robustness. The network emulation is controlled via the Network Emulation Interface from the Network Emulation Engine of the TEE. Run-time behaviour changes are made by the Execution Runtime Emulation and assure the identification of potential SLA violations. The strict isolation of the SUT within the sandbox is realised by encapsulating interfaces to external web services or IoT resources. Therefore, the service description mandatorily includes the IOPE description to all external services or IoT resources. Nevertheless, functional descriptions from interfaces only partly describe the utilisation flow of the interface (e.g, typically time deltas between request, typical response delays). To overcome this limitation the inclusion of a capture and reply mechanism is intended in order to reuse real communication with IoT resources and inject the traffic back in the test mode.

VI. Prototype Implementation

The requirements of IoT business services raise a lot of open test issues as mentioned earlier. In order to explain some of our concepts by example we have implemented parts of our proposed architecture to highlight the complexity of a (semi-) automated test case creation and execution approach. In the outlined example test cases are designed for our Context Provisioning Middleware [11]. The test case execution validates if a lookup service interface is working properly (integration tests). The Context Broker is requested via a HTTPS/Get interface. The correct answer is expected to be encoded in an Extensible Markup Language (XML) based language called ContextML [12] described within a provided XML Schema Definition (XSD) file.

As outlined in the previous section, the Test Design Engine (TDE) prepares the test cases for execution. After recognising the new service the TDE identifies an appropriate codec for HTTP and XML. Hence, knowledge from the defined structure of the XML data is transferred into a codec and the constraints of the provided XSD file are utilised for building the expected data structures in the TTCN-3 format [13]. An element of the related structures is shown in Listing 1. Basically, it enables casting the received data stream into the expected XML structure. Based on the XSD description the possible data structure consists of a *scopeEl* element that consists of a list of *par* elements structured in a *parA* element. In addition, the expected data types and data items are described. The next step is to build the test cases and the test execution flow. Even a very simple example like this can illustrate the required complexity of an automated process. Like many other data types, the XSD description allows interlaced structures without limitations of the length. Hence, it has to be tested against dynamic data structures. The templates depicted in Listing 2 are utilised to test the interface against the expected data input. As shown in Listing 2, the template restricts the data item ‘n‘ to the string ‘scope‘ and allows only letters from a-Z and numbers of 1 to 15. But where does this knowledge originate from? Data constraints like these are not included in the XSD description. Therefore, the services needs an additional semantic description as proposed in IV-B.

```
type record scopeEl {
  record {
    XSDAUX.string n,
    record of record {
      XSDAUX.string n,
      XSDAUX.string content
    } par optional
  } parA
}
```

Listing 1. XML Structure Described with TTCN-3

The templates are utilised by a test execution function. A control structure defines the flow of the test execution. The test execution function sends a response to the SUT and

```
template scopeEl.parA.par[-] parTest := {n := "
    scope", content := pattern "[a-zA-Z]#(1,15)"};
template contextML responseCheck(template scopeEl.
    parA.par p_list) := {
  content := {
    scopeEls := {
      scopeEl := {
        {
          parA := {
            n := "scopes",
            par :=  p_list
...
}
template GETInfo getInfoAuth := {...}
```

Listing 2. TTCN-3 Templates

 ISBN: 978-1-61208-233-2

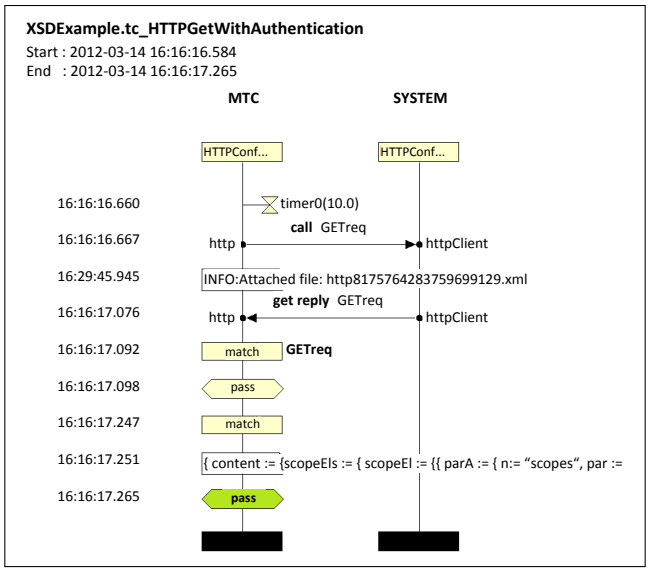


Figure 8. Test Execution Screenshot

validates the response with the shown template. Figure 8 shows the resulting graphical view of the the test execution realised with a tool called TTworkbench [14]. It shows the interaction flow between the Main Test Component (MTC) and the System to test. After receiving the *get reply* the test case is marked as *pass*.

VII. Conclusion and Future Work

In this paper, we discussed that the domain and application boundaries for IoT can be overcome with a business oriented service composition. Our life cycle management takes into account that IoT enabled services need to cope with the abstraction of heterogeneity, reliability and robustness by integrating (semi-) automated self-testing capabilities. Our model based testing approach addresses these issues and identifies a semantic test description enabling reasoning of test behaviour and suitability in the different phases of a service life cycle. An appropriate test architecture has been presented. Practical problems are discussed based on an IoT based service with a typical web interfaces. Due to the broad scope of the paper, the discussions are rather concentrated on a high level and future work work will include a more detailed description of the proposed approach, which appears to be a promising way utilising SOA in the IoT domain.

VIII. Acknowledgement

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n° 257521.

References

[1] R. Tönjes, E. S. Reetz, K. Moessner, and P. M. Barnaghi, "A test-driven approach for life cycle management of internet of things enabled services," in *Procedings of Future Network and Mobile Summit, Berlin, Germany*, pp. 1–8, 2012.

[2] I. Schieferdecker, Z. Dai, J. Grabowski, and A. Rennoch, "The uml 2.0 testing profile and its relation to ttcn-3," *Testing of Communicating Systems*, pp. 609–609, 2003.

[3] ETSI, "The testing and test control notation version 3 (ttcn-3)." European Standard 201 874, 2002/2003.

[4] Y. Cheon and G. Leavens, "A simple and practical approach to unit testing: The jml and junit way," *ECOOP 2002 Object-Oriented Programming, Springer*, pp. 1789–1901, 2006.

[5] M. Huo, J. Verner, L. Zhu, and M. Babar, "Software quality and agile methods," in *Proceedings of the 28th Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pp. 520–525, IEEE, 2004.

[6] W. Chengjun, "Applying pattern oriented software engineering to web service development," in *Procedings of International Seminar on Future Information Technology and Management Engineering, 2008. FITME'08.*, pp. 214–217, IEEE, 2008.

[7] G. Canfora and M. Di Penta, "Service-oriented architectures testing: A survey," *Software Engineering, Springer*, pp. 78–105, 2009.

[8] M. Presser, P. Barnaghi, M. Eurich, and C. Villalonga, "The sensei project: integrating the physical world with the digital world of the network of the future," *Communications Magazine, IEEE*, vol. 47, no. 4, pp. 1–4, 2009.

[9] W. Wang, S. De, R. Toenjes, E. Reetz, and K. Moessner, "A comprehensive ontology for knowledge representation in the internet of things," in *Procedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 1793–1798, IEEE, 2012.

[10] W3C, "Owl-s: Semantic markup for web services." W3C Member Submission 2004. Available online at http://www.w3.org/Submission/OWL-S retrived: October, 2012.

[11] M. Knappmeyer, N. Baker, S. Liaquat, and R. Tönjes, "A context provisioning framework to support pervasive and ubiquitous applications," in *Proceedings of the 4th European Conference on Smart Sensing and Context (EuroSSC)*, (Berlin, Heidelberg), pp. 93–106, Springer-Verlag, 2009.

[12] M. Knappmeyer, S. Kiani, C. Frà, B. Moltchanov, and N. Baker, "Contextml: a light-weight context representation and context management schema," in *Procedings of 5th IEEE International Symposium on Wireless Pervasive Computing (ISWPC)*, pp. 367–372, IEEE, 2010.

[13] I. Schieferdecker and B. Stepien, "Automated testing of xml/soap based web services," in *Kommunikation in Verteilten Systemen*, pp. 43–54, 2003.

[14] Testing Technologies, "TTworkbench." Website. Available online at http://www.testingtech.com/products/ttworkbench.php retrieved: October, 2012.

 ISBN: 978-1-61208-233-2

AndroLIFT: A Tool for Android Application Life Cycles

Dominik Franke*, Tobias Royé†, and Stefan Kowalewski‡
Embedded Software Laboratory
Ahornstraße 55, 52074 Aachen, Germany
{*franke, †roye, ‡kowalewski}@embedded.rwth-aachen.de

***Abstract*—The states and state transitions of mobile applications - often referred to as application life cycle - play a crucial role in high quality applications. An incorrect life cycle implementation might lead to unexpected application behavior and data loss. However, yet there are no tools available for supporting developers to implement the application life cycle correctly and to test application life cycle-related properties. This work presents the integrated tool AndroLIFT, consisting of two parts, for supporting the correct implementation of Android application life cycles. One part supports implementing and allows monitoring of application life cycles, even of multiple applications being in different states. The second part implements a unit-based testing approach, providing the possibility to test life cycle-related properties. AndroLIFT is implemented as an Eclipse plug-in to be integrated with the Android Developer Tools.**

***Keywords-application life cycle*; *unit-based testing*; *development tools*; *software quality*; *Android*.**

I. INTRODUCTION

Application life cycles describe the different process related states and state transitions of an application. Figure 1 presents the life cycle of an Android 2.2 Activity. An *Activity* is an Android application, which has a graphical user interface (unlike services). An Activity can be in one of four states:

- It is *shut down*, if it was not started, yet, or if it has been destroyed. The Activity holds no data in RAM.
- An Activity is *stopped*, if it is not visible to the user, e.g., another Activity currently holds the user focus.
- In the state *paused* the application might still be visible to the user, but it does not hold the user focus, e.g., an *incoming call*-dialog covers a part of the user interface.
- If an Activity is *running*, it usually is in foreground and holds the user focus. We show in [1] that this is not always the case. On Android, at each moment in time only one Activity can be in this state.

The two states s_1 and s_2 are intermediate states, in which the application never remains for a long period of time. The transitions are labeled with various method names, e.g., `onCreate()` and `onStart()`. These methods are callback methods, triggered by the Android system in case of a state change. But, not all state changes cause the execution of callback methods. The transitions labeled with *kill* mark state changes, in which the application is killed by the Android system, without invocation of any callback methods. Reasons for such killing might be lacking resources or an application crash. In this cases, the application has no possibility to react on a state change. But, to react on regular state changes, the developer can override the corresponding callback methods.

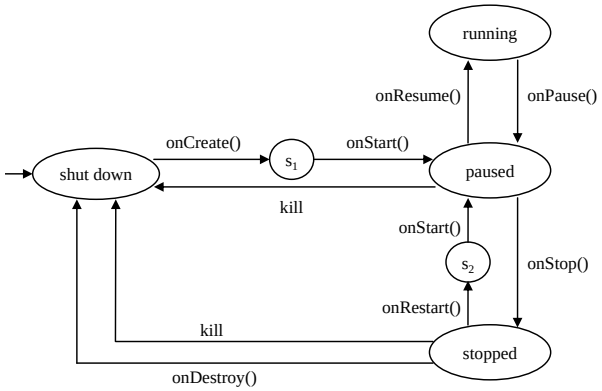


Figure 1. Android Activity Life Cycle [1]

Due to restricted resources and limited input/output capabilities, usually, modern mobile platforms, like Android, iOS and WP 7, have only one active user interface application running (plus some background services). This policy requires a special kind of scheduling (see Fig. 2). Each time a new GUI-application shall be opened, the currently running application first has to be stopped. For instance, on Android the running application first changes its state from *running* to *paused*. Then, the new application changes its state from *shut down* to *paused* and then *running*. Next, the other paused application is stopped and remains in this state. During this application-switch, already multiple life cycle callback methods are called (see Fig. 2). In each of the callback methods the application might have to turn off/on connections, hardware modules (e.g., Bluetooth, Wi-Fi, GPS, ...) or store data. An incorrect or insufficient implementation of the application life cycle might lead to unexpected application behavior and thus to bad user experience, poor usability and data loss [2], [3]. For instance, we pretend that *application A* in Fig. 2 makes use of the GPS module. It releases the module in `onStop()`, since the developer assumes that *application A* is first stopped,

 ISBN: 978-1-61208-233-2

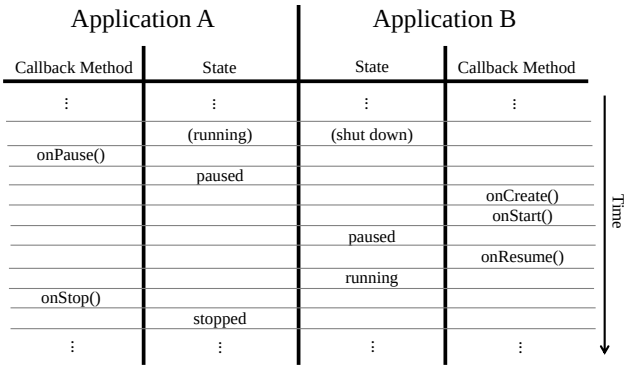


Figure 2. Scheduling two Applications on Android [4]

before *application B* is started. But, Fig. 2, which is the true scheduling on Android 2.2 [4], shows that *application B* is running before *application A* is stopped. If *application B* wants to access the GPS module when starting, it will fail, since *application A* still uses the module. After *application A* releases the module in `onStop()`, no life cycle callback methods of *application B* are called. The GPS module remains unused and *application B* does not use it, except if *application B* would actively poll for it, which is no life cycle action.

There are no tools available, yet, to help developers to find this certain kind of errors, implement the application life cycle or test life cycle-related properties of mobile applications. In this paper, we present *AndroLIFT*, a tool which helps developers of mobile applications to analyze, implement and test application life cycles. AndroLIFT is implemented for Android applications, but the concepts behind it can be implemented for other mobile platforms, too. We chose Android for a first implementation, since it is currently the most widespread mobile platform. Additionally, Android itself and the developer tools for Android are available open source. This allows to integrate AndroLIFT as a plug-in into the available Android Eclipse framework. Neither iOS nor Windows Phone, two competing mobile platforms, are available open source. The first functionality of AndroLIFT we present, supports the implementation of the application life cycle by allowing to monitor application life cycles during runtime in a life cycle view. It also eases the implementation of the life cycle, e.g., overriding the life cycle callback methods, by connecting a graphical view of the application life cycle to the source code editor. The second functionality integrates the life cycle testing approach from [4] into the Android Eclipse plug-in. It provides a user-friendly graphical interface to the life cycle testing library, eases implementation of the test approach, and connects it directly to the life cycle view.

The paper is structured as follows: Section II presents details about some Android developer tools, of which AndroLIFT makes use of. In Section III, the life cycle view is introduced. This view is extended by the testing functionality in Section IV. Section V concludes this work.

II. Background

This Section introduces the tools on which the AndroLIFT library and plug-in are based on. First, the Android Logcat tool, a logging tool for Android devices, as part of the Android SDK, is presented. Second, a brief description of the Android Development Tools, for development of Android applications with Eclipse, is given.

A. Android SDK, ADB and Logcat

The *Android Software Development Kit (SDK)* is a bundle of various tools, applications and documentation to develop software for the Android platform [5]. Since the Android SDK is a very rich bundle, but we only need few of those components for this work, we do not explain too much about the Android SDK itself. Therefore, we refer to the official Android SDK references.

One of the core components of the Android SDK for application development is the *Android Debug Bridge (ADB)*. ADB is a command line tool, which allows to communicate between a development machine and an Android device or emulator. For instance, it provides the possibility to send data to a device, remotely install and remove applications and forward ports. It also allows to receive log information from the device using the *Logcat*-tool. Android's Logcat allows on a development machine to view debug output from an emulator or connected device. It is the main logging mechanism on Android. Next to log messages sent by applications using the *android.util.Log*-class, it also provides various system information, as stack traces in case of an error and kill information, if a process is killed.

Logcat has a structured way of logging. For instance, each log information can be attached with certain information. Printing a log information with priority *debug* looks from the perspective of the developer as follows:

```
Log.d("MyTestClass",
      "Connection to server failed.");
```

The corresponding Logcat-output looks like:

```
D/MyTestClass( 1633):
       Connection to server failed.
```

We use this Logcat tool to send information about the life cycle state of an application to AndroLIFT.

B. Android Development Tools

The *Android Development Tools (ADT)* is a tool collection for development of Android applications with the Eclipse IDE. The ADT Eclipse plug-in extends Eclipse with different features, like Android projects, building and debugging Android applications, SDK tools integration, Android XML editor and integrated Android framework documentation [6].

 ISBN: 978-1-61208-233-2

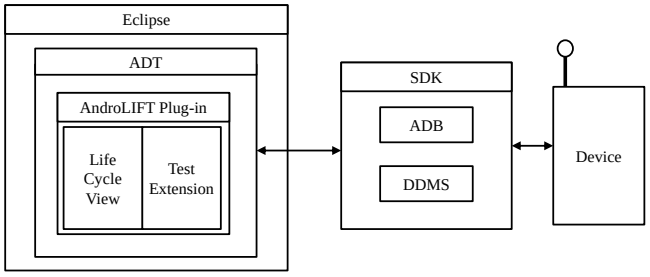


Figure 3. Integration of AndroLIFT into ADT

It is the common way to develop Android applications with ADT and Eclipse.

As these tools are available open source, we build our AndroLIFT plug-in on top of ADT. The advantage is that Android developers used to Android development with ADT get an integrated approach in a well-known environment. The integration of AndroLIFT into Eclipse and ADT is sketched in Fig. 3. AndroLIFT is fully integrated into the ADT environment. It consists of two parts, introduced in Sec. III and IV. ADT uses the Android SDK tools to communicate with devices. For the following work, especially the SDK-tools ADB and *Dalvik Debug Monitor Server (DDMS)* are important. DDMS allows further debugging services, like radio state information, screen capture on device and incoming call spoofing [7]. It usually connects to ADB to provide its full functionality. So, following the architecture in Fig. 3, the AndroLIFT plug-in never communicates directly with the device, but uses the same communication channels as ADT does: over ADB and DDMS.

III. Life Cycle View

To find out in which life cycle state an application is, the developer has to override the corresponding life cycle callback methods and print the corresponding information. There exists no other way to get this information on all mobile platforms like Android and iOS. But, as mentioned above, the life cycle of a mobile application is an important component for high quality applications. A graphical representation of the life cycle would help to examine the life cycle, its behavior during state changes and corresponding callback methods to react on certain events. We present such a view for the Android platform as part of ADT.

Figure 4 shows a screenshot of the AndroLIFT life cycle view of an Android Activity. The life cycle is taken from a reverse engineering approach [1], which is shown in Fig. 1. The intermediate states are left out (see Fig. 1), since for the developer it is only important what the resulting callback sequence on the corresponding transition path is, e.g., `onCreate()` is followed by `onStart()`. As a first step the developer has to specify on the left side of the life cycle view, which Activity of which package shall be monitored. Therefore the Activity can be executed on the Android emulator or a USB-connected real device. Then the current state of this Activity is marked by a dashed line. For instance, the Activity in Fig. 4 is currently in the state *shut down*. Next to the different states of the Activity, all available callback methods and kill-transitions are displayed as labels of the state transitions. If the monitored Activity changes its state, e.g., from *shut down* over *paused* to *running*, the corresponding path and states in the view are animated. To make the state changes comprehensible and traceable for the developers, the animations do not run in real-time, but slightly delayed. First, the transition labeled `onCreate()`, `onStart()` would be marked, followed by the state *paused*, transition labeled `onResume()` and finally state *running*, each marked for 500ms. Additionally, each call of a callback method is logged in the life cycle callback history view, presented on the left side in Fig. 4. It prints the name of the Activitiy (important in case of multiple running Activities), name of the callback method executed in that Activity and a timestamp of the call, to understand the order of the callback methods. With such a view on the application life cycle, the developer easily can find out the different state changes as a consequence of a certain event, e.g., incoming call or SMS.

If the developer knows and sees how his application behaves during runtime regarding its life cycle and which callback methods are called, he can easily implement a correct life cycle behavior. To ease the implementation of the life cycle, we added another feature to the life cycle view. By right-clicking a callback method, a menu pops up, with which the developer immediately can jump to the corresponding callback method in the source code editor. An example is given in Fig. 5, where the developer is about to modify the life cycle callback method `onStart()`. Additionally, if the corresponding callback method is not overridden, yet, AndroLIFT automatically overrides the callback method and places the cursor to the correct position in the source code editor. The developer immediately can start implementing the life cycle behavior.

Since the Android system itself, as all other modern mobile platforms, do not give any information about the current state, the application under test has to do so. It has to report the state of its life cycle to the life cycle view, each time the state changes. This information is needed by the life cycle view to trace the life cycle during runtime. This can be done in two different ways. One way is to extend an Activity class, called *DebugActivity*, which AndroLIFT provides. This class has already all code, which is needed by the life cycle view, encapsulated. This includes the initialization of the connection between the AndroLIFT plug-in and the application under test. It also automatically forwards state information to AndroLIFT via Logcat. If the developer does not want to use the DebugActivity, the code has to be placed manually in the corresponding applications, which is no big effort, either. For the `onPause()`-method the injected code

 ISBN: 978-1-61208-233-2

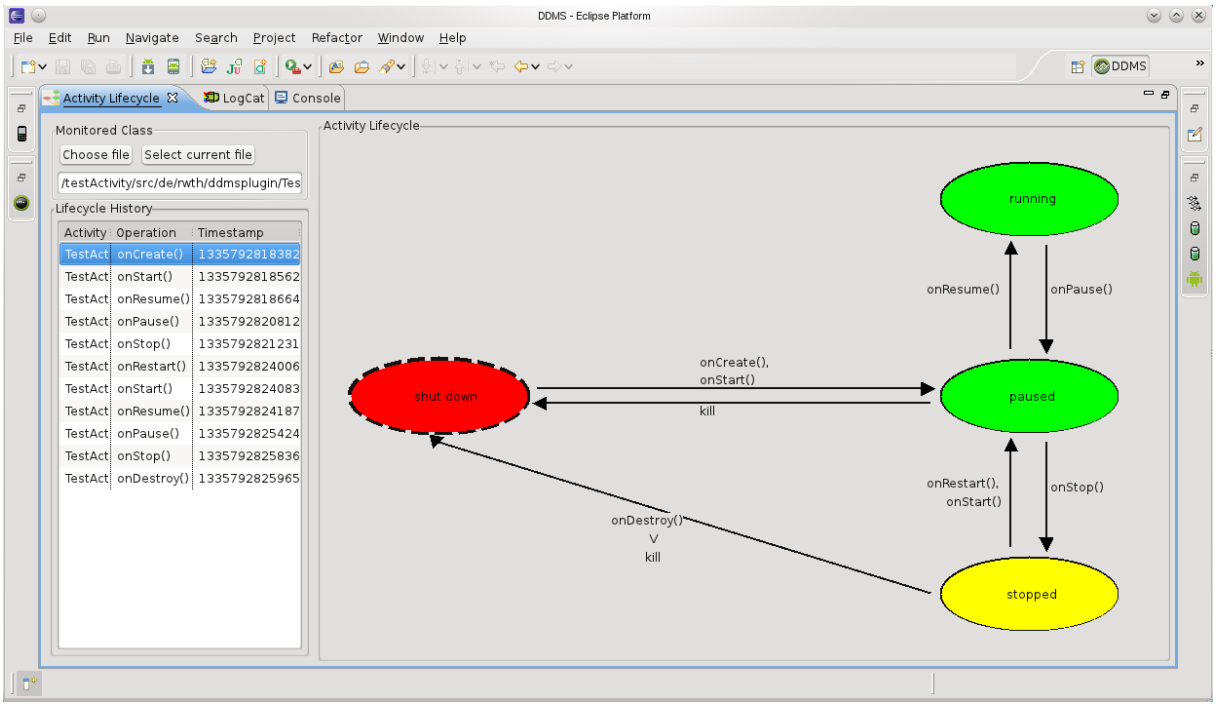


Figure 4. The Life Cycle View allows to monitor the Life Cycle nearly in Real-time

looks as follows:

```
Log.d(this.getClass.getName(),
    "onPause() called.");
```

The corresponding callback to the super class is mandatory in callback methods on Android and for AndroLIFT certain information containing the names of the package, Activity and life cycle callback method have to be printed to the Logcat tool. This has to be done for each life cycle callback method. This information is fetched by the life cycle view and processed accordingly. For instance, by knowing, which life cycle callback method was recently called, AndroLIFT knows the current state of the application.

IV. Life Cycle Testing

In a previous work [4], we present an approach to test life cycle-related properties on mobile platforms. This unit-based testing approach sees one Activity as a unit, regarding its life cycle. Life cycle state changes on modern mobile platforms are usually only triggered by the underlying system, not directly by the application itself or by another application [8]. For instance, if an application requests to be paused, the underlying system decides if and when to pause the application. In this sense a mobile application is separated regarding its life cycle, and thus in this sense a unit. This is not unit-testing in the original sense, where smallest testable part is separated and tested [9]. On Android, we see one Activity as a unit. We trigger the environment, e.g., making an incoming call to the Android system, and observe the reactions of the Activity regarding its life cycle.

To specify the expected behavior of an application we use assertions. An assertion can be derived from a part of the specification of the application, e.g., *if the user receives an incoming call while writing an e-mail, do not loose the already composed e-mail text*. On Android, if the user is writing an e-mail, the corresponding application must be in the state *running* (see Fig. 1). We know from [1] that an incoming call causes an Android Activity to be paused. Thus, following our testing approach, in `onPause()` assertions need to be defined, which store the current content of the affected text fields (subject, main text, etc.) and a reference to the text fields. The object reference is needed to be able to check those text fields after resuming the application. After the call, the application resumes again, which means that the callback method `onResume()` is invoked (see Fig. 1). So the previously defined assertions have to be checked in `onResume()`. The stored text is compared to the current text in the affected text fields and the test results are printed to the user. For more detailed information on testing life cycles of mobile applications, we refer to [4].

We implemented this approach for Android as a library, called *AndroLIFT runtime assertion library*. The package structure of this library is sketched in Fig. 6. Due to reasons of clarity, not all classes are displayed in this figure. The *LCAssertions*-class is the main class of the library. It handles, stores and checks all assertions. The *Util*-package

 ISBN: 978-1-61208-233-2

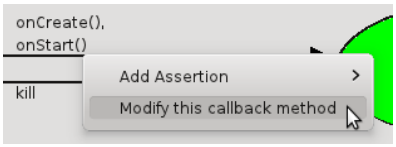


Figure 5. The Life Cycle View assists in implementing Life Cycle Behavior

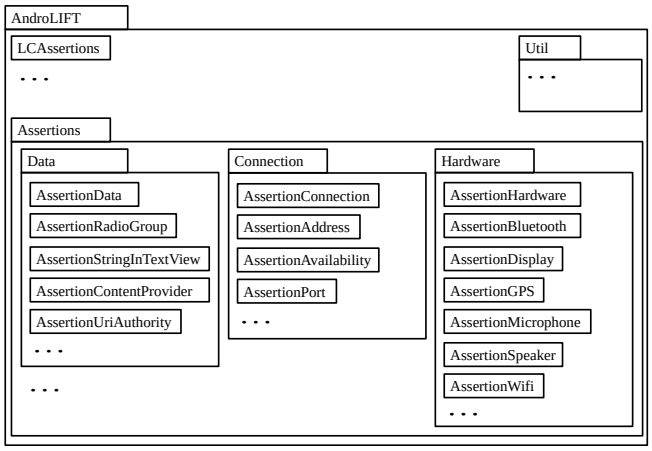


Figure 6. Package Overview of the AndroLIFT Runtime Assertion Library

holds different utilities, like database schemas for storing assertions. The *Assertions*-package contains various types of assertions:

Data Assertions: All assertions regarding data persistence, e.g., content of text fields, choice of radio buttons and list selection.

Connection Assertions: Assertions for checking if some kind of connection, e.g., IP/TCP or Bluetooth, is still available and active.

Hardware Assertions: Assertions about the status of hardware components, e.g., checking if GPS or Bluetooth is on.

The developer uses the different types of assertions to specify requirements to life cycle-related properties of his application and passes them to *LCAssertions*. This core class of the library manages the definitions and checks of the assertions depending on the current application state. By knowing the current state, AndroLIFT is able to check all assertions in the corresponding states. Since mobile applications might be killed, e.g., due to lacking resources, the library is also able to store assertions persistently. Therefore it uses the storage possibilities of the application under test, like the database on Android. With this concept, the developer can also check requirements like:

> *The application might not loose the content of an e-mail, even if it is killed by the system due to low battery.*

The corresponding assertions are then stored in the database, which is a persistent storage. After the corresponding test case is executed, e.g., low battery is simulated with the Android emulator, and the application is returned, the assertions can be restored from the database and checked.

The following code presents an example usage of the AndroLIFT library:

```
@Override
public void onPause(){
  super.onPause();
  Assertion a = Factory.
    createDataAssertion(STRING_IN_VIEW,
      textView1);
  androLift.assertThat(ON_RESTART, a);
  androLift.onPause();
}
```

With the help of an assertions-factory, the developer defines an assertion *a* about a string value in a text view object *textView1*. The library fetches the current text from the text view and stores it automatically. Further, the developer specifies that *a* shall be checked in the callback method `onRestart()`. The last line in this example tells AndroLIFT that `onPause()` is called, so the state changes to *paused*. Due to the restricted possibilities to get information about the current application state on Android, AndroLIFT needs to get this information from the application under test. So just like with the life cycle view (see Sec. III), the application under test needs to make a call to AndroLIFT in each life cycle callback method. Regarding the example above, for the callback method `onPause()` it is the line `androLift.onPause()`.

On top of this life cycle testing API, we developed a graphical user interface, which we integrated as an extension to the life cycle view plug-in. In this case there are various advantages of a graphical user interface over a library: The library itself was not integrated into the well-known ADT. Since the life cycle view is integrated into these tools, the testing extension is. The usability of the testing library is enhanced by this integrated solution. Further it is easier to learn and more intuitive to use than on code-level with corresponding code-level documentation.

With the graphical test extension, the user can create an assertion by right-clicking the corresponding life cycle callback method, in which the assertion shall be checked. With only few clicks the user is able to create the same assertion as given in the code above. First, he has to decide, which type of assertion he wants to define. Second, he needs to specify, where the object, e.g., a text view, is defined, since on Android user interface objects can be defined in Java as well as in XML. Finally, the developer needs to define the method, in which the assertions shall be defined. It will be checked in the method he right-clicked before. Figure 7 shows the output-view of the AndroLIFT test-extension. On the left side the developer can choose, which

 ISBN: 978-1-61208-233-2

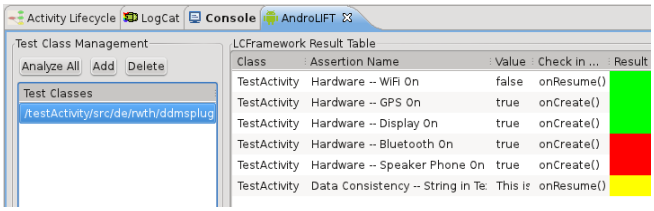


Figure 7. Output of the AndroLIFT Plug-in

test results from which Activity he wants to see. If his application contains multiple Activities, they are listed on the left side. After clicking on one of the listed Activities, the corresponding test results are printed in the table. The developer can see the type of the assertion, the attached value as well as the result. If the assertion passed, the result field is filled green. If an assertion did not pass, the result field is printed red. If the assertion has not yet been checked, e.g., since only the defining but not yet the checking callback method has been executed, the result field prints yellow. This way of reading test results is far more user friendly and less error prone than checking test results in a log file, as with the pure AndroLIFT runtime assertion library.

V. Conclusion

Application life cycles play an important role in the area of mobile applications. An incorrect or insufficient implementation of the life cycle might cause unexpected behavior of the application, leading to bad usability and even data loss. Until now, there are no tools available to analyze and test application life cycles.

In this paper, we presented AndroLIFT, a tool which helps the developer to monitor the life cycle, assists him in implementing it and testing life cycle-related properties. AndroLIFT is written as an extension to the ADT, the common way of developing Android applications with the Eclipse IDE. With the life cycle view the developer can observe and analyze the life cycle of his Android application. Developers easily learn about the behavior of the application life cycle to certain triggers, like an incoming call, and with which callback methods one can react appropriately. Further, with right-clicking the corresponding life cycle callback methods in the life cycle view he can quickly implement assisted the life cycle of his application. With the test extension of the AndroLIFT plug-in the developer has a user-friendly way to use the AndroLIFT testing library. From within the life cycle view he can create life cycle assertions, using the corresponding GUI. During and after test execution, e.g., simulation of an incoming call, the test extension of the life cycle view presents the results in a well-readable way, aligned to the well-known JUnit-testing tools.

By helping to learn and understand the life cycle of Android applications quicker and better, developers get a good feeling for the behavior of the life cycle to certain events. They can immediately see, with which life cycle callback methods they can react appropriately to certain life cycle triggers. With the test extension the developer can specify test cases by creating assertions in an intuitive and user-friendly way. The tool handles automatically all source code injections (except a few), which are necessary for working with the AndroLIFT runtime assertion library. Additionally, the life cycle test results are presented in a human readable and comprehensible way.

Acknowledgment

This work was supported by the UMIC Research Centre, RWTH Aachen University, Germany.

References

[1] D. Franke, C. Elsemann, and S. Weise, Carsten Kowalewski, “Reverse engineering of mobile application lifecycles,” in *18th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2011, pp. 283 – 292.

[2] D. Franke, S. Kowalewski, and C. Weise, “A mobile software quality model,” in *12th International Conference on Quality Software (QSIC)*. IEEE Computer Society, 2012, pp. 1 – 4.

[3] D. Franke and C. Weise, “Providing a software quality framework for testing of mobile applications,” in *4th International Conference on Software Testing Verification and Validation (ICST), Berlin, Germany*. IEEE Computer Society, 2011, pp. 431–434.

[4] D. Franke, S. Kowalewski, C. Weise, and N. Prakobkosol, “Testing conformance of lifecycle-dependent properties of mobile applications,” in *5th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2012, pp. 241 – 250.

[5] S. Komatineni and D. MacLean, *Pro Android 4*. Apress, 2012, vol. 1.

[6] R. Meier, *Professional Android 2 Application Development*. John Wiley & Sons, 2010, vol. 2.

[7] E. Burnette, *Unlocking Android: A Developer’s Guide*. Manning Publications, 2009, vol. 2.

[8] D. Franke, C. Elsemann, and S. Kowalewski, “Reverse engineering and testing service life cycles of mobile platforms,” in *2nd DEXA Workshop on Information Systems for Situation Awareness and Situation Management (ISSASiM)*. IEEE Computer Society, 2012, pp. 16 – 20.

[9] A. Hunt and D. Thomas, *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003, vol. 1.

 ISBN: 978-1-61208-233-2

Experiences in Test Automation for Multi-Client System with Social Media Backend

Tuomas Kekkonen, Teemu Kanstrén, Jouni Heikkinen
VTT Technical Research Centre of Finland
Oulu, Finland
{*tuomas.kekkonen, teemu.kanstren, jouni.heikkinen*}*@vtt.fi*

***Abstract*—Effective testing of modern software-intensive systems requires different forms of test automation. This can be implemented using different types of techniques, with different requirements for their application. Each technique has a different cost associated and can address different types of needs and provide its own benefits. In this paper, we describe our experiences in implementing test automation for a multi-client application with a social media backend. As a first option, traditional scripting tools were used to test different aspects of the system. In this case, the test cases were manually defined using an underlying scripting framework to provide a degree of automation for test execution and some abstraction for test description. As a second option, a model-based testing tool was used to generate test cases that could be executed by a test harness. In this case, a generic model of the behaviour was defined at a higher abstraction level and from this large numbers of test cases were automatically generated, which were then executed by a scripting framework. We describe the benefits, costs, and other properties we observed between the two different approaches in our case.**

***Keywords*-model-based testing; test automation; performance testing; data validation testing; web service testing.**

I. INTRODUCTION

Testing is commonly referred to as one of the most time consuming parts of the overall development and maintenance process of a software intensive system. To address this, various techniques have been developed to make test automation more efficient, each with their own costs and benefits. In this paper, we describe our experiences in implementing test automation for a multi-client system with a social-media backend. In implementing this system, two different approaches were applied to create and execute test cases with varying degrees of test automation. Both tested the system through its external interfaces built on top of HTTP requests with JSON data structures (REST style web services). We describe our observed costs, benefits, and limitations of each approach.

The first approach we applied was based on using existing test automation frameworks to provide a scripting platform for manually defining test cases and automating test execution. The tools applied are existing HTTP scripting tools to manually define test sequences for testing specific properties of the system. For us, the benefit with this approach is quick bootstrapping of the test automation process in using off-the-shelf tools, and the ability to manually define specific test cases for specific requirements. The cost is in creating large sets of test cases manually, which quickly becomes labour intensive and exhaustive. The main person responsible for this approach was the first author of this paper.

The second approach we applied was based on model-based testing (MBT). MBT is used to generate test cases from a model describing the system. This model typically describes the behaviour of the system in a form of a state-machine at a suitable abstraction level for generating the required test cases. The MBT tools provide components and modelling notations to make the modelling easier, and algorithms to generate test cases from the models. The test logic and integration with the test setup are domain specific and need to be created separately for each tested system. For us, the benefit with this approach is getting extensive coverage with automated test generation. The cost is in creating the models for test generation and integration with test execution. The main person responsible for this approach was the third author of this paper.

Guidance and coordination for both of these approaches was provided by the second author of this paper.

The rest of the paper is structured as follows. Section II presents the problem domain. Section III presents the manual test setup and experiences. Section IV presents the MBT test setup and experiences. Section V discusses the overall experiences in a broader context. Finally, conclusions end the paper.

II. BACKGROUND

We consider the system under test (SUT) here as a form of a web-application, where the different components communicate over HTTP requests. The service also provides a native mobile client interface and a social-media web-browser interface. However, in the testing phases described in this paper, we were interested mainly in testing the backend service. This is because the concerns of the project parties were on the high bandwidth and data processing requirements set by the data collection and transfer. Thus the user-interface part is not discussed in detail at this point, but only for the relevant interface and data processing parts related to these interfaces.

Often in web application testing the main goal is to get assurance that the service can handle all the user requests without problems. Therefore, it can be seen as performance testing. This requires assessing various properties such as

 ISBN: 978-1-61208-233-2

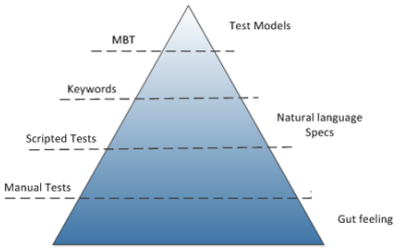


Figure 1. Test Automation Pyramid.

response latency, varying event sequences, event frequency handling, and error handling(as described e.g., in [1]). To estimate the capabilities the tester has to have some idea how many users the service will have and what type of requests there will be. The variance in effect to the server between requests can be in processing load, I/O load and network load. Based on the estimates the service is loaded with different requests and then maybe bottlenecks are discovered. Then the development team can optimize those weaknesses in the service. Usually, some big faults in the server configuration or server side scripting are discovered at this phase. In our experience, basic properties such as database query implementations and even chosen image formats can cause serious performance issues depending on choices made in design and development. Besides basic verification of the basic functionality of the SUT, our goal was also to verify the performance of the system and identify any bottlenecks.

We view test automation as a form of a pyramid, where manual testing is at the bottom and MBT at the top. This is illustrated in Figure 1. In this pyramid, the different levels of test automation build on top of the lower levels, and in our experience, it is not possible to implement the higher levels effectively without first having the lower levels working. This is similar to, for example, how Blackburn et al. described evolution of test automation from basic test execution to scripting based on action words, and finally model-based testing at the fourth generation [2].

Manual test automation at the lowest level can be just a user clicking on controls of a graphical user-interface and observing how they feel it should behave. Test scripts are typically a form of computer program written to perform a specific sequence. Keywords are abstractions that are transformed into test scripts by a test automation framework, allowing one to create test cases using higher level language concepts. As MBT generates test cases, optimally it should be able to generate them in terms of higher level elements such as keywords to avoid having to put low level details in to the test model. Keywords are a suitable approach for this as they are already supported by several test automation frameworks. An example of such integration can be found, for example, in [3]. An approach where the MBT tool can also be guided through embedded keywords, effectively combining benefits of both approaches can be found in [4].

In terms of a web service such as the one we tested here, this type of an effort is not directly possible, but rather scripting tools are required for even the most basic testing, where the reference for expected behaviour is typically the natural language specification of the SUT. The ability to execute test scripts is also a prerequisite for MBT. For this reason, the MBT part also requires having the same underlying execution platforms available as the manual scripting part we described as the first approach applied in our case study. The MBT approach also requires formalizing the specification as a suitable behavioural model from the typical natural language form. For this reason, we can only expect to have to spend more effort on the MBT approach. Thus, it is also important to understand the potential benefits to be able to evaluate where it may be most applicable and produce realistic gains.

Besides the choice of the level of test automation applied, other factors also affect the overall cost of the solution. This includes integration with other tools in the tool chain such as test management tools, defect tracking, continuous integration, virtual machines and others (see e.g., [5], [6] for examples). However, these are common requirements for any level of test automation and as such we focus here on the parts specific to test automation itself, where the differences are greater.

A. Previous Experiences

Test automation has been considered one of the biggest cost factors in the software development process for a long time. For example, Persson and Yilmazturk describe establishing test automation as a high risk and high investment project in their experience report [7]. They also list 32 pitfalls encountered in taking test automation into use in practice. These are too numerous to list all here, but some of the most relevant ones include poor decision making with regards to what is automated and to what extent, considering full test automation as a replacement for manual testing, and the misconception that test automation would always lead to savings in labour costs. We provide in this paper some added insights into these pitfalls and information to help make more informed decisions on what, where, and how to automate.

A similar experience report is also provided by Berner et al. [8]. Among other things, their experience shows misplaced expectations for fast return of investment of test automation, limited test automation beyond test execution, wrong abstraction levels for tests. They also note again that it is also their experience that automated testing cannot replace manual testing, but should rather be seen to complement it. Similarly, importance of proper maintenance is also emphasized. Again, we provide in our case study information on our experiences in manual vs. highly automated testing

 ISBN: 978-1-61208-233-2

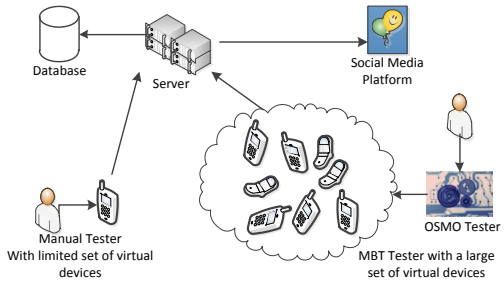


Figure 2. Description of the target service and the testing methods.

approaches (from the lower levels of the pyramid to the higher levels).

Several case studies on using MBT have also been published. These typically focus on presenting the benefits achieved, while providing little discussion or comparison with a manual test automation approach for the same SUT. Examples of such studies include MBT for the Android smartphone platform [9], for the healthcare domain [10], automotive domain [11], information systems [12], and several others. In this paper, we describe not only the benefits of MBT as a separate study, but a comparison with manual testing for the same project. Both testing methods aimed for the same goal, but ended up using a slightly different approach.

B. Our Testing Domain

The SUT here is a multi-user service with mobile phone clients and a backend server. A diagram of the environment is presented in Figure 2. The backend server also hosts web applications for social media purposes. The mobile client is a standalone application that collects information about the user behaviour and provides a view for the user to this information on the mobile device. The user can also activate a social media component, in which case the data is uploaded to the backend server once a day. When activated, the backend server processes the data and provides a summary of information through a social media application.

Data collected by the mobile phone application consists of logged events with timestamps and usage information of different applications on the phone. The data is XML formatted, but also contains comma separated data fields within.

The backend service receives the data sent by the mobile clients and stores it in temporary data storage. After a certain time, it is batch processed to a more detailed form. It is then stored in the final location to be used by the social media application. The triggering of the batch process causes high spikes in the service load. Due to the heterogeneous user set, the social media also causes high load in the form of complex database queries with varying parameters based on different user configurations and their associated social network graph.

The desired capacity of the service was calculated to be around 100 000 data uploads per day and the same amount of social media content requests per day. The registration message, which is only sent once per user, is not included in the estimates. Numbers were calculated from the estimated user count of 1 million from which one tenth were expected to enable data upload to the backend server, and little less were expected to also use the social media application. The service was expected to have users globally so the load is distributed evenly around the day. Little higher service load was expected to occur in the European daytime as the biggest set of users were expected to be from Europe. Size of the data was estimated to be 4 KB for every data upload. These numbers were safe estimates to leave some room for error. This would simply calculate into 400 megabytes of data each day and roughly 1 upload per second.

The estimates are hard to make for such a service with different users and mobile phone capabilities. Also, the popularity of the social media application varies between countries and its usage is hard to estimate. Therefore, the distribution of request types sets the problem for estimating the capabilities of the service and therefore the testing of the service. As it is a web service it faces the same problems as any other service with multiple users. An estimate has to be made as to what kind of requests can be expected in what ratio and in what sequence. If the more data intensive requests are more common, then the required capabilities of the service are different than in a situation when it mostly faces computing load and only little bandwidth load.

To create test scenarios to test this type of service, a tester needs to write complicated test scripts. For example, a tester could write scripts that create requests to the server and have the script repeat these requests any number of times. This script could also include a mechanism to observe if the system responded properly and if the data gets processed into final storage properly. Checking the data from the storage requires a secondary access point to the database. Last step would be to make sure everything gets done properly no matter what the sequence of actions is.

The input in testing was chosen to contain the most sensitive and data intensive requests types. We will call them request1 and request2. Request1 contained user registration information with many details about the user and the mobile device. Request2 contained certain logged information collected during a day from the phone. This data was naturally linked to the previously created user.

III. Manual Testing

During the main development phase of the system, the manual test approach was the one first applied to create a limited set of required test scenarios. Here, by manual, we mean simple scripting based testing. It was seen as a means to quickly achieve the needed coverage for the most critical requirements.

 ISBN: 978-1-61208-233-2

Apache Bench[13], Siege[14], Grinder[15] and Curl[16] were used as tools for the testing in this phase. The goal was to verify the core functionality of the service, and to see how many requests it can serve in a certain time.

A. Process

1) Create desired requests: Using Curl, a simple HTTP request program, the tester first created a request and verified that it gets appropriate response. Variables in the request were static.

2) Implement the request into test program: When the request type is clear and the parameters are correct, this request can be implemented as a test case for Apache Bench, Siege or Grinder. Both were used in this test and they provided similar features for this type of simple testing. Apache Bench cannot be used as versatilely as Grinder. Grinder provides the possibility to define test case with multiple requests which is useful in this scenario. Apache Bench was only used to test performance with one type of request.

3) Scripting: Apache Bench runs a single HTTP request with defined amount of threads for defined amount of times. To run repeatedly with different parameters in repeats and concurrence the tester has to implement a small shell script to make this process easier for repetition purposes too. The same applies for Siege and they are both strictly command line operated.

4) Execute the tests: Executing here means letting the test script repeat the test with desired configuration. In Siege and apache bench the testing can be configured by changing the amount of simultaneous requests and interval in requests. The goal was to reach a certain performance in terms of requests per second with any reasonable count of simultaneous requests. In practice, we targeted 50 to 200 simultaneous requests. Simultaneous users or requests here mean executing the HTTP requests as fast as possible with parallel threads. The same number of simultaneous users in a web page consumes fewer resources than this because users do not repeat their requests that fast.

5) Observe performance: Documenting and analysing the results is the final part of testing. Server performance was analysed with one type of request at first. This way, some sorts of estimates of the performance were determined and some rough limitations could be set. In reality, other variables and requests can have great impact on performance.

B. Weaknesses

During the initial testing phase many weaknesses were noticed in the manual testing process. The test scripts were not able to produce variance in the test data. This was partially solved by creating random variables in the test data. However, it still did not produce suitable data when certain type of data was needed in different scenarios. When the test script is not designed from the beginning to be functional enough with variables and randomness, its configurability is weak.

IV. Model-based testing setup

MBT is often used to help testers increase test coverage. This can be achieved by varying the test sequences or variables in test steps through the usage of the model. In our test environment we have used the OSMO Tester MBT tool described in [17].

MBT and test automation in general are processes that require time for setting up and implementing. Their advantage is usually observed in a longer running software development project.

In our case, we used only MBT with the goal of fully integrating all the required external testing tools into the MBT tool. This way, the test generator could directly generate and execute test steps against the SUT. This way, the test engineer does not need to set up complex execution environments for different tests, but can run them at once. The tools used were Grinder, OSMO and JDBC SQL connector. To integrate this environment the solution was to import OSMO as a library to Grinder Jython based environment and use the test generator from there. Similarly it was also possible to use JDBC connector for the database connection.

A. Process

1) Grinder: The first phase of building the MBT setup was setting up Grinder with the requests and response verification. The main goal was to see how Grinder works and how it could be integrated with OSMO to perform the testing against SUT. Here, the input data, along with user and request count configuration, was specified inside Grinder.

2) Grinder and OSMO: In the second part of the process, the OSMO was brought into the setup. The possible requests and responses were included in the OSMO system test model. This model was used in Grinder Jython script which then produced input data for the Grinder HTTP requests. This way, the requests had increased coverage while still being able to verify the responses.

3) Grinder, OSMO and SQL: The last part of the MBT setup was bringing the database element into testing. Naturally to verify the server operation the database processors correct operation had to be confirmed. Bringing this into the test automation really improved the testing process in our case. In manual testing, verification was limited to checking one or few requests ending up correctly into the database after processing. In the MBT setup, the effect of every request was checked against the database.

B. Weaknesses

As mentioned many times in reports about model-based testing, the launching and covering the requirements takes lot of time in the beginning. For us, also the matter of learning the environrment delayed the process of implementing model-based testing. A decision about where to

 ISBN: 978-1-61208-233-2

implement the model had to made. We decided to create a Java class including the model which was then brought to the Jython based environment of Grinder. This setting up phase also lead us to broaden the requirements that we would cover with the setup. It felt the limited set of testing requirements set by the project was not worth implementing as a model-based setup. Therefore, we went on to implement a more advanced performance testing setup which included the validation of performance in malicious and exceptional situations.

V. Discussion

The manual testing in the project was done by a test automation expert who had a long history with the project and its previous iteration. He also had experience with web applications and the tools used for manual testing prior to the manual test effort. He was able to get going and implement the first test cases in a matter of hours. Further test cases were implemented over time and were similarly low effort. However, they were limited to testing only the core features with as few test scripts as possible. The manual testing reached higher coverage in terms of covering all the features of the service. Despite this it lacked in covering those requests with different types of parameters.

The MBT testing in the project was done by a test automation expert who had his background in a different domain, was unfamiliar with the SUT, and with the tools used to implement model-based testing for the SUT. Learning the tools and the expected behaviour of the SUT took several months of effort to build the different iterations of the MBT solution described in Section IV. Each iteration took about one man-month of effort in this case. The MBT implementation of the testing covered the request types in the web service that are most resource intensive. The main goal was to provide more variance in the performance testing which was the main concern in deploying of the service.

A. Method evaluation

Evaluating the difference between processes is difficult in a case such as this when the two in comparison have different components to start up with. The person performing the manual testing had different background than the person building the model-based testing. Therefore, the efficiency and performance of the processes are hard to evaluate, but we concentrate on the performance and usability of the test environments which came out as a result.

1) Setting up: As stated earlier the difference in setting up these two types of test environments is noticeable. Model-based testing takes more time in early stages, but saves time in longer run. This is often shown in research about model-based testing[18]. This effect was clearly visible, especially in the great effort of setting up the model-based environment in contrast to ease of making the manual tests run accordingly.

2) Repeatability: Repeating the created set of requests was not an issue in manual testing. When a set of tests and scripts were done, repeating those and changing the user and request count was easy. In this sense manual testing can easily execute the test cases as a form of regression testing. This is important, especially when something is changed in the SUT and there is need to test the performance for possible improvements. MBT can cause issues here if care is not taken to make the generated tests repeatable. If the test cases are regenerated from a changed model or from the same model with different parameters, the test contents can change and the results are not consistent. Because of this care has to be taken to document which model configuration or which set of generated tests was used for which reported performance.

3) Modification: At one point in manual testing there was a certain set of requests that repeated in many test cases, but in a different sequence. When a new request type was added it had to be added to every test case, which was very laborious. Making this type of change in the MBT setup does not require this type of effort. The tester only needs to add the request type into the model and generate new tests with varying sequences and payloads.

4) Sequence direction: In manual testing, the tester has to create the sequence by hand and this repeats through the whole test run. This is true if Siege is used to define the test case requests. In the MBT approach we used sequence direction for example by defining that request1 has to repeat n times before any request2 type requests are generated. Support for rules like these makes it easy to produce guided variance in test generation.

5) Request type distribution: This part was the most difficult for manual testing with the tools we used. Siege was the tool that was able to take the requests as a list and therefore with some added manual effort it was possible to modify the ratio of request1 and request2. Still this was not that flexible. OSMO provides the possibility to simply give a weight to each step in the test case and this way the ratio of each request can be defined to a great accuracy in a long test case. Different variant combinations can also be easily generated to any numbers automatically with OSMO Tester.

6) Payload configuration: The biggest advantage of MBT in this scenario was the ease of modifying the content of the requests. This was illustrated earlier in Figure 2. It was possible to make each request have different type of input data and to include some faulty data. Even though this might not be essential in performance testing, it is important to know whether certain type of data causes performance issues or lockups.

VI. Conclusion

We have described here a set of experiences in implementing both manual and model-based performance testing for a single project. The results show how manual testing was an

 ISBN: 978-1-61208-233-2

effective way to bootstrap the testing process and produce focused test cases for the system under test. With good knowledge of a large set of suitable tools for the domain, it is also possible to create a good initial test suite with reasonable effort.

However, maintenance of large test suites quickly becomes laborous, and addressing extensive variation in testing manually is too expensive. This is where MBT can help. MBT can have a significant initial investment required, but can result in much easier means to evolve a test suite and to address large scale variation requirements. Familiarity with the tools, techniques and the domain can also work to significantly reduce the initial investment required.

In the end, we can say that for us the best process would be to start with manual testing and when the system under test and the test environment are stable enough bring in model-based testing. At this point, MBT can also be applied to address some of the needs for manual testing using specific configuration properties of OSMO Tester such as those described in [4].

We continue to apply MBT to new projects with similar and different properties, collect the experiences and improve our understanding of how the different types of testing may benefit the testing process in different contexts. We wish to have a case to properly apply and analyze both manual and model-based approach. In this case the model-based approach was added later then whe possibility appeared. This way, the starting points and motives of the testing were not exactly same. With a dedicated case we could provide a better comparison of the types with metrics and cost analysis. In the beginning of each process the testing had same goal. However when model-based testing advanced, its purpose was altered a little to cover slightly different objective.

REFERENCES

[1] A. Shahrokni and R. Feldt, "Robustest : Towards a framework for automated testing of robustness in software," in *3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 78–83.

[2] M. Blackburn, R. Busser, and A. Nauman, "Why model-based test automation is different and what you should know to get started," in *International Conference on Practical Software Quality and Testing*, 2004, pp. 212–232.

[3] T. Pajunen, T. Takala, and M. Katara, "Model-based testing with a general purpose keyword-driven test automation framework," in *4th IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 242–251.

[4] T. Kanstén and O.-P. Puolitaival, "Using built-in domain-specific modeling support to guide model-based test generation," in *7th Workshop on Model-Based Testing (MBT 2011)*, 2012.

[5] B. Peischl, R. Ramler, T. Ziebermayr, S. Mohacsi, and C. Preschern, "Requirements and solutions for tool integration in software test automation," in *3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 71–77.

[6] V. Safronau and V. Turlo, "Dealing with challenges of automating test execution architecture proposal for automated testing control system based on integration of testing tools," in *3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 14–20.

[7] C. Persson and N. Yilmazturk, "Establishment of automated regression testing at abb: Industrial experience report on avoiding the pitfalls.," in *19th International Conference on Automated Software Engineering (ASE'04)*, 2004, pp. 112–121.

[8] S. Berner, R. Weber, and R. Keller, "Observations and lessons learned from automated testing," in *27th International Conference on Software Engineering (ICSE'05)*, 2005, pp. 571–579.

[9] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based gui testing of an android application," in *4th IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, 2011, pp. 377–386.

[10] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella, and B. Hasling, "Applying model-based testing to healthcare products: Preliminary experiences," in *30th International Conference on Software Engineering, (ICSE 2008)*, 2008, pp. 392–401.

[11] E. Bringman and A. Krmer, "Model-based testing of automotive systems," in *3rd International Conference on Software Testing, Verification, and Validation (ICST 2008)*, 2008, pp. 485–493.

[12] P. Santos-Neto, R. Resende, and C. Pádua, "An evaluation of a model-based testing method for method for information systems," in *ACM Symposium on Applied Computing*, 2008, pp. 770–776.

[13] "Apache bench, the apache software foundation," http://httpd.apache.org/docs/2.0/programs/ab.html, 2012, [retrieved: June-2012].

[14] "Siege load tester home page," http://www.joedog.org/siege-home/, 2012, [retrieved: June-2012].

[15] "The grinder, a java load testing framework," http://grinder.sourceforge.net/, 2012, [retrieved: June-2012].

[16] "cURL command line tool," http://curl.haxx.se, 2012, [retrieved: June-2012].

[17] T. Kanstén, O.-P. Puolitaival, and J. Perälä, "Modularization in model-based testing," in *3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 6–13.

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

 ISBN: 978-1-61208-233-2

Project in Control: An Innovative Approach

Jos van Rooyen
Bartosz ICT BV
Arnhem, the Netherlands
jos.van.rooyen@bartosz.nl

***Abstract*-Nowadays, companies are still struggling to execute and deliver IT-projects successfully. Several reasons can be mentioned. The main question, however, is how the business can gain confidence in the new or adapted IT-system? This article describes an approach where, from a business point of view, the IT-system is monitored to assure that the business can use the IT-system in its daily operations. The approach is developed in practice, during several projects over the last 5 years. The experiences are collected and structured in such a way, that projects and companies can apply the method into their own organization. All the companies who applied the approach were successfully Ready for Shipment.**

Keywords-Quality Monitoring; Change Management; Integrality; Ready for Shipment and Practical Based Approach.

I. INTRODUCTION

Nowadays, companies are still struggling to execute and deliver IT-projects successfully. Often, requirements are not met, business operations are poorly prepared and the business processes are not supported well by the delivered IT-system. IT and business are not aligned, project's deadlines are far from planned and the budget is exceeded significantly. No wonder new projects are welcomed with skepticism.

The causes of failed projects are all recognized and nevertheless the IT industry is still struggling with this issue and apparently not able to change this. Is it possible to change this at all? How to gain more control to successfully implement an IT-system? How to avoid decrease of quality when the time pressure on the project increases? How to ensure that the end-users are well prepared, accept the new system and actually experience added value?

This article describes an approach to improve the success rate of IT projects. Instead of focusing on IT, in this approach, the business processes are leading and taken as a starting point. From there, it is derived how it can be supported automated or manually and how that together affects the organization. The approach is not the ultimate solution, but the experience till now is that by applying the described approach, the success rate of the IT-projects will increase significantly. How to achieve this? By not looking at IT solely! The approach that will be described is a Practical-Based Approach. The approach was developed in practice during several projects over the last 5 years. Table I shows the number of projects, the domain where the approach is applied and the size of the projects. The experiences are collected and structured in such a way, that projects and companies can apply the method into their own organization.

The paper has the following structure. Section II describes the cause of failure of IT-projects. Section III describes the integral approach. Section IV presents the application of the approach. Section V concerns the related work. Finally, in Section VI, conclusions and future work are mentioned.

II. CAUSE OF FAILURE IT-PROJECTS

A much referred cause is the shaky base of the project. The business case is not specific enough [8]. Requirements are incomplete, ambiguous or even unclear [1][10][11][12][13]. A more soft cause is the alignment of business and IT [4]. The business is not understood by the IT department and vice versa. How can a system be developed, if you do not know what process will be supported or by whom it will be used?

Another cause is the skill of the project member [9]. Despite the fact that a lot of methods, processes and techniques have been developed, the quality of the individual skills determines the end result. The system development process is lengthening. Many projects are, e.g., outsourced to low-wage countries. As such, this does not have to be a problem; but, it complicates communication because of the distance and different languages it brings cultural differences and, as stated before, results in wrong products. If the IT-project is not sure what it wants, how to expect that others deliver the right product? A well known example is the annotation of numbers. Are you talking about inches or cm?

One final cause to be mentioned here is the one-sided way of looking to projects. Very often the technology is leading. High tech solutions and state-of-the-art are the miracle words and triggers. Developers tend to forget for whom they are developing software and in what context their contribution is used. It is obvious that there is no fit as long as it is not considered and treated in coherence along, with the to be supported processes and the organization for which it is meant for.

Despite the fact that project management methods, development methods, techniques, development

 ISBN: 978-1-61208-233-2

environments etcetera are improving enormously and expanding continuously, this does not seem to result in more successful projects. On the contrary, from publications it is derived that the percentage of successful projects hardly exceeds 45% [1].

Considering all the above, there is no single cause for the failed projects. One thing is for sure. You cannot blame the IT only [7]. The business does not know exactly what they want, they are not responsive enough, rely too much on others such as vendors and bring in new requirements as the project is already underway [2][3][5]. The processes should be taken as a starting point. What is required for example to implement a procurement process? A new system itself is not enough. What about the users, workflow, offices and communication? Herein lays the core causes of the problems that occur. IT should not be looked at solely from an IT-perspective, but from a business perspectives instead and, as an integral part of the triangle: IT, Processes and Organization.

The question is: "How to solve this?" The answer is not straightforward. Having made mistakes in the past and having learnt from them, an integral approach has been developed, where elements of different fields and skills are applied and combined. Fields such as, Change Management, Testing and Quality Assurance. Elements from the fields Testing and Quality Assurance have been clustered under the header of Quality Monitoring. The application of the combination of the elements from different fields ensures that projects can be implemented more successful. An integral approach, in which from the business perspective to look at IT and the consequences for the organization, has been proven to be a successful one. Herein lays the unique character of the approach. The approach has been developed over the last few years during various projects and gradually evolved to what it is today. One thing is for sure, the development of this approach will continue for years.

III. THE INTEGRAL APPROACH

The distinctive character of this approach, is by looking from an integral point of view to the required business processes, the required resources (IT) and, the (future) organization. The integral approach is based on two main components, i.e., a base architecture and a 5 steps action plan. The base architecture is presented in Figure 1.

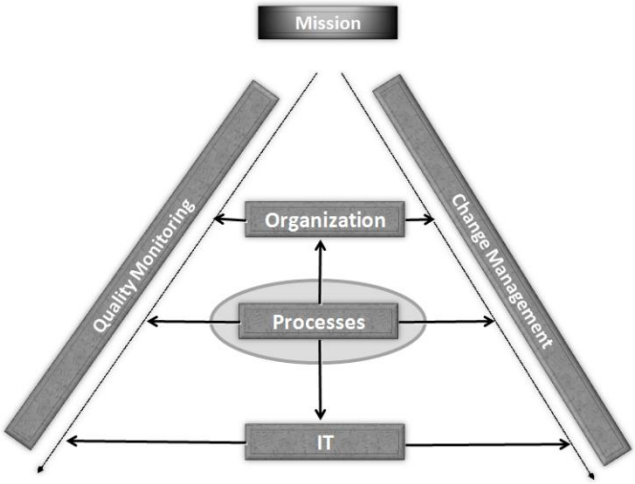


Figure 1. Base Architecture

In the end, the approach should provide enough confidence in the IT-system and organizational readiness to decide that the IT-system can be released at a certain moment in time. In such a way, that you know when the system is released, the planned activities can be continued, insight in risks is provided, knowledge about the weaknesses is present, goals as defined in the business case have been reached and assurance to the organization is achieved. This way, you can maintain focus during the project.

Every project starts with a certain goal, preferably derived from the business goals as defined by the organization [8]. The project should contribute to that goal. Often, these goals are derived from the mission statement of the organization. The goals are elaborated in a business case to a project's objective. From this objective, the main focus is determined. This could be changes of the business processes, the functionality of the information system, changes to the business or even a combination of these three.

In this approach, the business process is always the starting point. From here, the needed changes in IT and subsequently the consequences for the organization are derived. These insights are the base for defining the Change Management plan. The input gathered from this approach, is also used in defining the Quality Monitoring plan.

In order to be able to apply this approach in sequential steps, the 5 steps action plan has been developed. This roadmap will guide organizations from business case to a fine tuned implementation of an IT-system.

The 5 steps action plan

The 5 steps action plan consists, as the name already suggests, of 5 sequential steps that contribute to a fine tuned implementation. The 5 steps action plan is shown in Figure 2. A short description of the 5 steps is given below.

 ISBN: 978-1-61208-233-2

Step1: Visioning

Visioning is the preparation of a successful transition, in which on forehand the consistency between processes, IT and organization, is defined, to sustain implementation and embedding of the information system.

Step 2: Reconnaissance

During the reconnaissance step the scope of the implementation and the embedding of the information system in relation to processes, IT and organization, in consistency with the vision, is explored. The purpose of this step is to get a clear picture of the goal of the project. What material is available, what kind of development process is used and who are the most important stakeholders?

Step 3: Commitment

During the commitment step, vision and reconnaissance will be elaborated into a commitment agreement (contract, plan, quality monitor plan, change plan). This is the blueprint for the implementation and embedding of the information system.

Step 4: Realizing

Realization consists of developing, implementing and embedding of the information system in the organization, according the agreed quality level over the axes of processes, IT and organization. By observations, it might be necessary to adjust vision, reconnaissance or agreement.

Figure 2. 5 steps action plan

Step 5: Improving

In the step of improving, the effect [14][15] of the implementation, will be evaluated and if needed, processes, IT and organization will be optimized. To do so, the Deming circle: plan, do, check and act, can be applied.

The 5 steps action plan assumes that the steps are taken sequentially. This is correct, but on basis of observations, one may need to take a step back. If it appears that the Change Management plan is not effective due to whatever reason, one should go back to the step reconnaissance and adjust the strategy.

IV. APPLICATION

From experience, the approach, as outlined, can be applied in all type of projects, like inhouse projects or offshore development (see Table I and further explanation in the next sections). This approach is not only applicable for new projects, but for releases as well. Depending on the targets, a large and solid process can be used, or a quick and pragmatic process. Independent of its size, it has been proven that the approach is suitable along with different development methods, like Waterfall [6], Agile [27], and Rational Unified Process (RUP) [28].

Keep in mind that a defined plan is not static. Depending on deviations, the plan must be adjusted accordingly and timely. One should not only regard the ideal path, but also regard the situation that deviations rise. For instance when requirements are not achieved as expected or not all defects are solved.

A. *Experiences*

The described method has been developed over the past years and evaluated against literature [17] [18][19][20][21][22]. In many projects, the approach has been applied and gradually shaped. At first an inventory of the current situation “as is” of the project was conducted. The problem that occurred was that it was difficult to determine whether the IT-system suffices the business needs. For that reason acceptance criteria were defined, including entry and exit criteria, requirements and product risks. Based on the acceptance criteria, it became possible to measure the quality of the IT-system and the Operational Readiness of the organization. Another major development was the idea to not only measure and monitor the IT-project and report findings, but also to cooperate in the improvement of all findings together with all involved parties such as business units, third party software vendors and system management.

Change management was able to made adjustments to their plan, based on the results of the quality monitoring activities. An example is the so-called known error. A known error is an accepted bug in the software of an IT-system for which a validated workaround is available. In that case, this bug will not affect the business. However

 ISBN: 978-1-61208-233-2

Change Management has to communicate this known error to the stakeholders.

This is an example how Quality Monitoring and Change Management amplified each other. Another angle of integrality. This way, the integral approach has been matured to the current level. The projects, where this approach has been applied, are all released successfully due to the continuous improvements.

It appeared that in environments in which a lot of (third) parties are involved, the approach worked very well. Business knew what was going to be delivered, what the quality would be, and whether the users and organization were ready to support the change and to use the product. By defining and agreeing the acceptance criteria up front, it was possible to review the test process of the several third party vendors. To determine the coverage degree for instance of the applied test sets.

It is about production readiness versus product readiness. Production readiness means that the organization is ready and prepared to use the new IT-system (product readiness). That includes cultural changes, migration of information, training of employees and measurement of satisfaction of customers.

B. Applicability

The applicability of the base architecture is high. The approach was applied into large international complex commercial organizations. Industry, banks and insurance companies, but also in large (semi) government organizations, the base architecture proved to be a big gain. The architecture is not only applicable in the IT domain, but we believe it is applicable on projects in general. However, there is no evidence yet. The experience till now is collected in IT or IT related companies [16]. Detailed information is gathered in Table I.

Table I. FACTS & FIGURES APPLIED PROJECTS

Domain	No. of projects	Type of project	Size in million euro's
Industry	2	Third party development	>20
Government	2	Third party development	>200
Energy & utilities	2	Third party development Inhouse development	>10
Semi government	1	Third party development	<1

The integral approach has appeared to be usable in a whole, but also parts of the approach can be used autonomic. Especially the step Realizing. Reason for this is that projects already started before we were involved into the projects.

C. Validation of the approach

As stated before the described integral approach is a practical based approach. The experience is that validation of the approach was hard to achieve. The approach, which was chosen, is also practical based. Based on the findings the approach was expanded with new techniques. Applying these techniques in the project the effectiveness could be measured. Based on these measurements, the approach was validated. In the situation the techniques were not sufficient enough; new ideas were developed to solve the findings. On this way, the approach was validated on a continuous base.

V. RELATED WORK

The presented integral approach is at the moment really unique in the industry. Existing approaches are focusing on IT solely [18][23]. The presented integral approach, focus not only on IT but also on the business processes and the related organization. Another main advantage is that the presented approach, not only look to the System Development Life Cycle but also to the implementation phase and the system management phase [26]. There are some interesting developments related to parts of the presented approach. These are focusing on product quality [24]. However, there is no interaction with the involved business processes and organization. Another development is around the topic of Quality Supervision [25]. This development is focusing on improvement of the total System Development Life Cycle. The goal of Quality Supervision is to remove all waste in the chain.

VI. CONCLUSIONS AND FUTURE WORK

This article described an integral practical based approach, from a business point of view, to collect information to determine if the organization is ready for usage of the new or adapted IT-product. The described approach is applied in several large IT-projects successfully (see Table I). All projects are released without major problems. Applying the presented approach has several advantages such as: Organization is ready for shipment, knowledge about weakspots is delivered and the organization is able to decide on a structured way to go live or not. One of the main topics for the upcoming period is to develop a structured questionnaire, which can be used in the reconnaissance step to determine the current situation. Based on the results, concrete steps can be defined. By executing a lot of projects in the coming years more experiences must be collected to validate the integral approach.

 ISBN: 978-1-61208-233-2

ACKNOWLEDGMENT

I like to thank my fellow authors, Jan Fokke Mulder, Hans Somers, Hanneke Kroon van der Linde and especially Jurgen van Amerongen. I would also like to thank all the companies who have provided input and used the method in their development process.

REFERENCES

[1] Ernst & Young, "ICT barometer," ict-barometer.nl/rapporten, 2009.

[2] K. Lindhout, "5 valkuilen bij veranderen *IT*," FM.NL, mei 2010.

[3] N. Beenker, "Opdrachtgever grootste risico bij IT-projecten," nicobeenker.nl, May 2010.

[4] R. Poels, "Beïnvloeden en meten van business-IT alignment," Amsterdam: Dissertation VU University, 2006.

[5] L. Dohmen, "Hoe adviseurs, coaches en goeroes onbewust verandering blokkeren," ManagementSite.nl, 2011.

[6] W. Turner, R. Langerhorst, G. Hice, H. Eilers, E. Remmerde, and A. Uijttenbroek, "SDM – System Development Methodology," Rijswijk: PANDATA, 1990.

[7] K. Buren, "Waarom mislukken al die IT projecten," Persberichten.com, May 2011.

[8] R. de Jong, M. Webster, A. Bouma, and A. de Jager, "Overheid gebaat bij business case," Automatiseringsgids, April 2011.

[9] T. Mulder and H. Mulder, "Kwaliteit projectteams onder de maat," Automatiseringsgids, Dec. 2011.

[10] R. Glass, *"The Software Research Crisis,"* IEEE Software 11(6): pp. 42-47, Nov. 1994.

[11] N. Fenton, S. Lawrence Pfleeger, and R. Glass, "*Science and Substance: A Challenge To Software engineers*," IEEE Software, pp. 86-95, July 1994.

[12] R. Charette, "*Why software fails,*" IEEE Spectrum, Sept. 2005.

[13] R. Glass, "Facts and Fallacies of Software Engineering," Boston: Addison-Wesley, 2010.

[14] R. van Solingen and E. Berghout, "*Goal Question Metrics,"* Berkshire: McGraw-Hill Publishing Company, 1999.

[15] B. Lohman and J. van Os, "*Praktisch lean management,*" Geldermalsen: Maj Engineering Publishing, 2010.

[16] J. van Rooyen, J. Mulder, H. Kroon vd Linde, H. Somers, and J. van Amerongen, "Project de Baas," Den Bosch: UTN Publishers, 2011.

[17] B. vd Burgt and I. Pinkster, "Succesvol Testmanagement: een integrale aanpak," Den Haag: ten Hagen & Stam, 2003.

[18] T. Koomen, L. van der Aalst, B. Broekman, and M. Vroon, "TMAP Next for result driven testing," Den Bosch: UTN Publishers, 2006.

[19] K. Jung and G. van de Looi, "100% succesvolle IT-projecten," Amsterdam: Pearson Education Benelux B.V., 2011.

[20] L. de Caluwe and H. Vermaak, "Leren Veranderen," Alphen aan den Rijn: Kluwer, 2006.

[21] S. Covey, "The Seven Habits of Highly Effective People," Free Press, 1989.

[22] P. Crosby, "Kwaliteit totaal," Deventer: Kluwer, 1993.

[23] C. Schotanus, "TestFrame," Den Haag: Academic Service, 2008.

[24] J. Hofmans and E. Pasmans, "Quality Level Management," Den Bosch: UTN Publishers, 2012.

[25] R. Marselis and E. Roodenrijs, "the PointZERO vision," Groningen: LINE UP boek en media bv, 2012.

[26] J. van Amerongen, "softwareapplicaties: goed gebouwd en toch niet af!?," TNN, pp. 35-39, Jan. 2012.

[27] K. Beck, "Manifesto for Agile Software Development," agilemanifesto.org, 2001.

[28] R. Collaris and E. Dekker, "RUP op maat," Den Haag: Academic Service, 2011.

 ISBN: 978-1-61208-233-2

From Model-based design to Real-Time Analysis

Yassine Ouhammou, Emmanuel Grolleau and Michael Richard
LIAS / ISAE-ENSMA
86961, Futuroscope, France
{*ouhammoy, grolleau, richardm*}*@ensma.fr*

Pascal Richard
LIAS / University of Poitiers
86961, Futuroscope, France
pascal.richard@univ-poitiers.fr

***Abstract*—Real-time and embedded systems are sharply impacted by wrong design choices detected at a very late stage of the life-cycle. This impact is often due to timing constraints which are related to structural and scheduling analysis capabilities. The timing constraints analysis requires an expertise in both design and scheduling analysis. In this paper, we highlight some temporal analysis related difficulties. In order to help designers and to improve the real-time system design to be corrected at an early stage, we propose an approach which is based on modeling oriented scheduling analysis.**

Keywords-model-based design; structure verification; scheduling analysis validation.

I. INTRODUCTION

The temporal validation of real-time systems is mainly based on the scheduling theory or model checking of the system using a formal model (timed Petri nets, timed automata, temporal logic, etc.). For example, the system analyst or expert system designer can analyze the temporal behavior of a set of tasks scheduled by a scheduling algorithm using algebraic methods, called feasibility tests, in order to prove that temporal constraints will be met at run-time.

The complexity of real-time systems leads to use model driven engineering which has gained more popularity among real-time system developers. Recently, the integration of scheduling analysis in a model driven engineering process has improved. On the one hand, several standardized design languages such as the UML-MARTE [1] or AADL [2], provide sets of non-functional properties for the specification, analysis and automated integration of real-time performance of critical distributed computer systems. On the other hand, many commercial and free schedulability analysis tools provide some scheduling analysis tests in order to help designers during the analysis phase such as MAST [3], Cheddar [4], etc. The meta-models of those timing analysis tools differ. Therefore, each of these tools uses a different set of concepts to create the input models for simulation and analysis. Despite all these standard design languages and these analysis tools, the use of these standards for scheduling analysis is still expensive in terms of design expertise.

In this paper, we propose a process assisting the designer step by step in order to verify that a system structure is coherent and respects all real-time architectural and behavioral rules. Once all structural rules are respected, we help designers to choose the feasibility tests corresponding to their design by extracting real-time information, analyzing it as a task model and then checking if the extracted information respects the task model assumptions of a third-party tool. Our approach is based on a decision tree.

The remainder of this article is organized as follows. The next section is an overview of real-time scheduling concerns and some related works. Section III introduces our global process, the relevant concepts and their utilizations. Section IV presents the analysis aspects of our approach. Finally, Section V summarizes and concludes this article.

II. BACKGROUND AND RELATED WORK

A real-time system is interacting with a physical process in order to insure a correct behaviour. For this, it is implemented as a set of parallel, interdependent functionalities. Parallelism is often ensured by the multitasking paradigm, relying on a operational layer offering task scheduling. Thus, the real-time problematic is based on three axes: the hardware equipments, the task models and the scheduling theory.

When validating a critical real-time system, starting from the real task system S, a task model S' which is a worst-case of the real system has to be chosen. Then, the worst-case behaviour of the task model S' is analyzed in an acceptable delay. All along this process, if the task model S' semantics is poor, then the way S is modeled leads to pessimistic feasibility analysis. In order to decrease the modeling gap between S and S', several advances have been made on the task models, like practical factors. As examples, the multiframe models which have been proposed for multimedia systems [5], serial communication systems use transaction models [6] [7], self-suspension tasks [8], precedence constraint anomalies [9], etc.

Real-time scheduling theory provides a set of scheduling algorithms and algebraic methods called feasibility tests. Scheduling theory has been originally studied for the basic Liu and Layland model [10], and extended to cover more advanced and precise task models. Usually, feasibility tests prove that a software model using a set of hardware resources will respect real-time requirements. The time complexity of the analysis is a very important point: as an

 ISBN: 978-1-61208-233-2

example, testing the schedulability of periodic independent tasks with an implicit deadline using a deadline-driven scheduler on a single processor is a very easy problem and requires a linear time in the number of tasks. A small change in the hardware, software or operational context has a big impact on the temporal analysis. For example, with the same hypothesis, if we consider a fixed-priority scheduling policy, in the case where the periodic tasks are either non strictly periodic, or strictly periodic and released simultaneously. Then, the feasibility problem is NP-hard in the weak sense (pseudo-polynomial feasibility test). But, if the tasks are strictly periodic and not released simultaneously, then the feasibility problem is NP-hard in the strong sense, while one could, for test efficiency reasons, choose to use a pseudo-polynomial feasibility test as a sufficient (but non-necessary) test for this problem.

In order to obtain a thin design granularity and to avoid the over-sizing of the hardware resources of critical systems, some research works have proposed several results in this major, such as the sensitivity analysis [11], [12], the priority assignment [13], and the multi-criteria optimization [14].

The architectural verification and temporal validation of real-time systems can be difficult for system designers. Recently some research works aim at helping designers during the modeling phase in order to get a coherent system architecture. Roquemaurel et al. [15] are interested just in the architectural validation to ensure the architecture coherence by using formal methods and constraint satisfaction problems. Plantec et al. [16] have proposed a panel of design patterns, once the system design respects design pattern rules then a scheduling analysis corresponding to this pattern can be be applied [16]. Peres et al. [17] have also proposed a verification method for real-time system by using model checking. These research studies focus on one kind of validation (architectural requirements or temporal requirements) of the real-time systems. Moreover, for those which are interested in real-time scheduling, they do not reduce the gap between the real system and the task model, nor offer the closest scheduling model when the real system does not correspond to an existing analyzable model.

We propose an approach based on model driven engineering in order to assist designers to validate their architecture during the design phase and then to facilitate the choice of the appropriate analysis during the design phase. This approach called MoSaRT (Modeling oriented Scheduling analysis of Real-Time systems) consists of a domain specific language enriched by a formal language. Our goal is not to compete with other design languages or to compete with existing scheduling analysis tools, but we suggest MoSaRT to cope with the modeling difficulty and to reduce the gap between the standardized real-time languages and temporal analysis tools.

III. Modeling oriented scheduling analysis

A. Brief description of MoSaRT language

MoSaRT framework is an intermediate framework between real-time design languages and temporal analysis tools. This framework is based on the MoSaRT language which is conceived as a domain specific modeling language for embedded real-time systems. It is based on four pillars: the platform, the behavior, the analysis and the functional model. These aspects are based on the notion of viewpoint complementarity. Each viewpoint represents a side of the global system (see Figure 1).

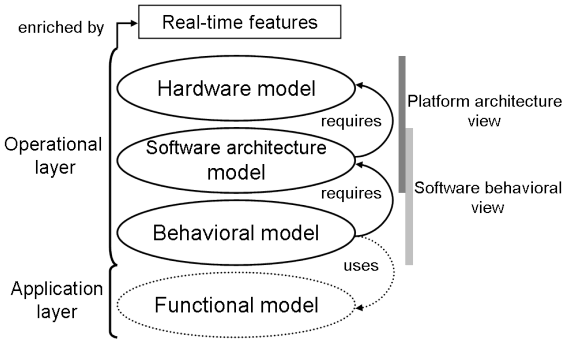


Figure 1. Different views of a real-time system designed through MoSaRT language

MoSaRT language is an instance of Ecore [18], which is a scripting language for meta-models. Moreover, Ecore is an implementation of the MOF (Model Object Facility) [19] under the integrated development environment Eclipse [www.eclipse.org] which is a well-equipped framework.

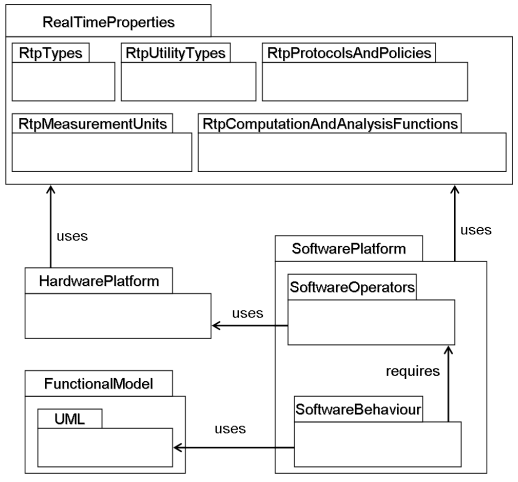


Figure 2. General structure of MoSaRT language

The architecture of MoSaRT language is organized in various packages and sub-packages. Figure 2 shows a global overview of the MoSaRT design language. For more details about MoSaRT language, readers can see related

 ISBN: 978-1-61208-233-2

papers [20], [21].

B. Formal semantics for MoSaRT language

The MoSaRT meta-model uses similar concepts as those used in the real-time system domain. For a good understandability of structural rules (Section IV-A), this section defines the main structural concept semantics and their inter-relationships without introducing the real-time properties.

Definition: Global System: In MoSaRT meta-model, the global system element "gs" represents the real-time system. It is characterized by the software platform S_{gs}, the hardware platform H_{gs} and the application F_{gs}. Then, gs = $<S_{gs}, H_{gs}, F_{gs}>$.

Definition: System's software platform: The software platform is composed of the software operators and the software behavior. MoSaRT language contains many software operators for modeling the tasks, the processes, the interaction resources (remote and local), etc. In order to ensure software operators consistency, especially tasks, behavioral elements offer to the designers the possibility to express the communication relationship, the precedence relationship, the trigger mode, etc. Thus: $S_{gs} = <S_O, S_B>$, where S_O is a subset of software operators set $\mathbb{SO}$, and S_B is a subset of software behavioral elements set $\mathbb{SB}$.

Definition: Software Operators: We define some mathematical notions that are used in this section:
- $\exists!$ means there exists exactly one
- f is a total function from the set E_A to the set E_B, if $\forall$ a $\in E_A$, $\exists!$ b $\in E_B$ such that f(a) = b
- h is a partial function from E_A to E_B, if A $\subset E_A$ and $\forall$ a $\in$ A, $\exists!$ b $\in E_B$ such that h(a) = b
- g is a total bijection from E_A to E_B, if and only if $\forall$ b $\in E_B$, $\exists!$ a $\in E_A$ with b = g(a)
- r is a relation from E_A to E_B, if A $\subset E_A$ and $\forall$ a $\in$ A, $\exists$ B $\subset E_B$ where r(a)= B
- $\oplus$ means "exclusive or"

We admit that:
- $\mathbb{LCR}$ is a set of Local Communication Resources
- $\mathbb{RCR}$ is a set of Remote Communication Resources
- $\mathbb{EMR}$ is a set of Exclusion Mutual Resources
- $\mathbb{TA}$ is a set of Task Activities that we define further

Then:

- $\mathbb{IR}$ = $\{ir_1, ir_2, ..., ir_n\}$ = $\mathbb{LCR} \cup \mathbb{RCR} \cup \mathbb{EMR}$, is a set of interaction resources, where each ir_i denotes an interaction resource, and $\mathbb{LCR}$, $\mathbb{RCR}$ and $\mathbb{EMR}$ are disjoint sets. Thus:
  $\forall$ $ir_i \in \mathbb{IR}$, $ir_i \in \mathbb{LCR} \oplus ir_i \in \mathbb{RCR} \oplus ir_i \in \mathbb{EMR}$
- $\mathbb{ST}$ = $\{st_1, st_2, ..., st_n\}$ is a set of Schedulable Tasks, each task is characterized by $st_i = <sp_j, \Omega_i, \Xi_i, \Psi_i, ta_i>$ where:
  - belongsTo is a total function defined as: $\forall$ $st_i \in \mathbb{ST}$, belongsTo(st_i) = sp_j, $sp_j \in \mathbb{SP}$
  - writesOn is a relation defined as: $\forall$ $st_i \in \mathbb{ST}$, writesOn(st_i) = Ω_i, $\Omega_i \subseteq \{ \mathbb{LCR} \cup \mathbb{RCR}\}$
  - readsFrom is a relation defined as: $\forall$ $st_i \in \mathbb{ST}$, readsFrom(st_i) = Ξ_i, $\Xi_i \subseteq \{ \mathbb{LCR} \cup \mathbb{RCR}\}$
  - accessesTo is a relation defined as: $\forall$ $st_i \in \mathbb{ST}$, accessesTo(st_i) = Ψ_i, $\Psi_i \subseteq \mathbb{EMR}$
  - representedBy is a total bijection defined as: $\forall$ $st_i \in \mathbb{ST}$, representedBy(st_i) = ta_i, $ta_i \in \mathbb{TA}$
- $\mathbb{SP}$ = $\{sp_1, sp_2, ..., sp_n\}$ is a set of Space Processes, each process is characterized by $sp_i = <T_i, sp_j>$ where:
  - $\forall$ $st_j \in T_i \subseteq \mathbb{ST} \Rightarrow$ belongsTo(st_j) = sp_i
  - inherits is a partial function defined as: $\forall$ $sp_i \in \mathbb{SP}$, inherits(sp_i) = sp_j, $sp_j \in \mathbb{SP}$

Thus, $\mathbb{SO} = \mathbb{IR} \cup \mathbb{ST} \cup \mathbb{SP}$ is a set of Software Operators.

Definition: Software Behavior: Let $\mathbb{TR}$ is a set of Triggers, and:

- $\mathbb{TA}$ = $\{ta_1, ta_2, ..., ta_n\}$ is a set of Task Activities, each task activity is defined as $ta_i = <st_i, tr_j, IA_i, OA_i, \Lambda_i>$ such that:
  - representedBy^{-1}(ta_i) = st_i, $st_i \in \mathbb{ST}$
  - triggredBy is a partial function defined as: $\forall$ $ta_i \in \mathbb{TA}$, triggredBy(ta_i) = tr_j, $tr_j \in \mathbb{TR}$
  - precededBy is a relation defined as: $\forall$ $ta_i \in \mathbb{TA}$, precededBy(ta_i) = IA_i, $IA_i \subseteq \mathbb{TA}$ and $\forall$ $ta_j \in OA_i \subseteq \mathbb{TA} \Rightarrow \exists$ $ta_i \in$ precededBy(ta_j)
  - containedBy is a total function defined as: $\Lambda_i \subseteq \mathbb{AS}$, $\forall$ $as_j \in \Lambda_i$, containedBy(as_j) = ta_i
- $\mathbb{AS}$ = $\{as_1, as_2, ..., as_n\}$ is a set of steps, each step is defined as $as_i = <\kappa_i, IS_i, OS_i>$ where:
  - $\kappa_i \in$ K = {action, acquire, release, send, receive, read, write} which is a set of step kinds.
  - stepPrecededBy is a relation defined as: $\forall$ $as_i \in \mathbb{AS}$, stepPrecededBy(as_i) = IS_i, $IS_i \subseteq \mathbb{AS}$ and $\forall$ $as_j \in OS_i \subseteq \mathbb{AS} \Rightarrow$ $as_i \in$ stepPrecededBy(as_j)

So, $\mathbb{SB} = \mathbb{TR} \cup \mathbb{TA} \cup \mathbb{AS}$ is a set of Software Behavioral elements.

In this subsection, we have shown just a part of some MoSaRT elements and their relationships. We have not introduced the real-time properties, nor how we relate the operational model to the functional model without any impact on the functional layer.

IV. Structural and scheduling analysis

In object-oriented modeling, graphical models are not sufficiently expressive to be able to express an entire precise specification. Moreover, it is often necessary to describe additional constraints on model instances. To specify these

constraints, formal languages have been developed. Generally, these languages use complex notations that require mathematical knowledge. Nevertheless, clarity and simplicity are among MoSaRT purposes. Moreover, MoSaRT can be used by different actors (designers and analysts) who are not necessarily proficient in different fields. Therefore, MoSaRT is enriched by several rules in different severity. It encapsulates these rules and generates just the error or information message which can be understood easily. For implementing structural and analysis rules, we have opted for OCL (Object Constraint Language) [22] for two reasons. The first one is the clarity of OCL, which is a formal language that is conceived to be read and written easily. The second reason is related to the meta-meta-model used, which is Ecore [18]. Ecore language operates very well with OCL and allows to define the constraints, the operations and the derived properties.

A. *MoSaRT structural rules*

In this section, we propose the different kinds of structural rules that must be respected in order to have a coherent design. We have separated structural rules into three groups. Architectural rules, vivacity rules ans safety rules. We treat these different kinds of rules through an example.

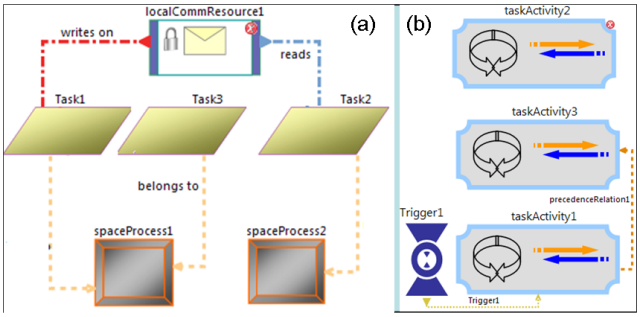


Figure 3. Checked models: Architectural rule (a) and vivacity rule (b) are violated

We consider a real-time system which contains two processes, a set of tasks and a set of interactions resources (see part (a) of Figure 3). The architectural constraints ensure the possibility of a good utilization of all elements that are modeled in the platform architecture. For instance, if *Task1* and *Task2* communicate through a local communication resource then *Task1* and *Task2* must belong to the same space process. This rule is formalized as follow:
$\forall$ $st_i \in \mathbb{ST}$ and $\forall$ $st_j \in \mathbb{ST}$
if $(\Omega_i \cup \Xi_i) \cap (\Omega_j \cup \Xi_j) \neq \{\}$
and $((\Omega_i \cup \Xi_i) \cap (\Omega_j \cup \Xi_j)) \subset \mathbb{LCR}$
then $sp_{st_i} = sp_{st_j}$

Listing 1 shows the implementation of this architectural rule as an OCL invariant.

Listing 1. An expressed architectural rule in OCL

```
invariant SoLocalCommResourceRule1:
(self.oclAsType(SoCommunicationResource).writerTasks
->union
(self.oclAsType(SoCommunicationResource).readerTasks))
->forAll(t1,t2 | t1 <> t2 implies t1.process = t2.process);
```

Consequently, the architecture that is shown in Figure 3 (part (a)) is not validated.

The vivacity rules ensure the correctness and completeness of the global behaviour. This kind of rules is very important especially for behavioral model. For example, in the behavioral model of real-time system designed with MoSaRT language, a scheduling activity must be triggered by a time trigger or external event trigger. Else, a scheduling activity must be preceded by another scheduling activity. This precedence relationship can be designed as a precedence synchronization or as communication relationship. So,
$\forall$ $ta_i \in \mathbb{TA}$
if $\nexists$ $tr_j \in \mathbb{TR}$ where triggredBy(ta_i) = tr_j
then precededBy(ta_i) $\neq \{\}$

By translating this rule to OCL, we obtain:

Listing 2. An expressed vivacity rule in OCL

```
invariant SbTaskActivityRule1:
self.oclAsType(SbSchedulingActivity).trigger
->isEmpty() implies self.oclAsType(SbSchedulingActivity).
inputSquencingRelation->notEmpty();
```

Therefore, *TaskActivity1* and *TaskActivity3* which are shown in Figure 3 (part (b)) respect the vivacity rule that is previously defined, contrariwise to *TaskActivity2*.

Safety rules guarantee that no erroneous behavior will happen especially when a designer enriches the model. For instance, if the objective of a designer is to have a detailed design, then the designer can specify the core of a task activity by adding steps.

Among MoSaRT safety rules, we can find the following one:
$\forall$ $as_i \in \mathbb{AS}$ if κ_i = *acquire*, then $as_i \notin$ *stepPrecededBy*(as_i)

This rule means that an acquire step can not precede itself. An "acquire step" means that task gets a semaphore in order to access to a critical shared resource. Then, the defined rule must be respect to ensure a safe system behavior. The Listing 3 shows the corresponding OCL rule.

Listing 3. An expressed safety rule in OCL

```
invariant SbStepPrecedenceRelationRule1:
self.sourceStep.oclIsTypeOf(SbAcquireStep)
implies not self.targetStep.oclIsTypeOf(SbAcquireStep);
```

In this section, we have shown the structural rules which must be respected during the design phase. Through MoSaRT, the validation of a real-time system structure is incremental. This approach has two advantages. The first one is to cope with scaling problems. So, the verification process stops a the first violation rule. The second advantage is to keep the traceability, then inform the designer about the model element that causes the invalidation of the system. In the next section, we expose the way MoSaRT rules detect

 ISBN: 978-1-61208-233-2

the ability of a real-time system to be analyzable and then, how it proposes the possible scheduling analysis which is matching with the analyzable system.

B. MoSaRT scheduling analysis ability rules

The purpose of MoSaRT language is not to provide scheduling analysis, but to help designers to verify their systems in order to conclude about the schedulability of their systems and to help in its dimensioning. MoSaRT contains several scheduling analysis rules which guide designers to choose the appropriate scheduling analysis tests. Moreover, MoSaRT can also offer to a designer several external scheduling analysis tools that provide these tests, such as Cheddar [4], MAST [3], etc.
Each analysis test depends on the model completion phase. For example, sensitivity analysis [11][12] is a dimensioning technique which can be solicited at an early design phase. Moreover, the model completion is measured by the kind of real-time properties mentioned in a real-time system design. For instance, a design which can be mapped to a Liu and Lalyland task model [10] could be analyzed as a transaction model [6] if offset-time properties were mentioned.

In this paper, we focus on scheduling analysis validation. MoSaRT offers a set of scheduling analysis ability rules in order to check the meta-model instance and then to infer the task model that corresponds to this instance. In the case where no task model corresponds to the designed instance, MoSaRT should suggest the closest task model. To facilitate the understanding about the manner MoSaRT detects scheduling ability of a design, we expose that trough a simple example. This example is based on a simple real-time system. It is composed of three independent periodic tasks which are executed on a processor using a fixed priority scheduling policy. Each task is characterized by a release date r_i, a worst-case execution time Ci, a relative deadline Di, a period Pi and a priority $Prio_i$ (the smallest value is the highest priority). Table I summarizes these characteristics.

Task	r_i	Ci	Di	Pi	$Prio_i$
Task1	0 ms	2 ms	4 ms	4 ms	1
Task2	0 ms	1 ms	4 ms	4 ms	2
Task3	0 ms	1 ms	8 ms	8 ms	3

Table I
VALUE OF TASK CHARACTERISTICS

Figure 4 shows the design of this example through MoSaRT language. This model respects all structural rules; then, we can apply schedulability analysis rules in order to know the possible analysis tests which can match this model.

The scheduling analysis ability rules are a set of assumptions. These assumptions are collected from different task models and they are implemented in MoSaRT language as a set of OCL rules that are not necessarily true. These OCL

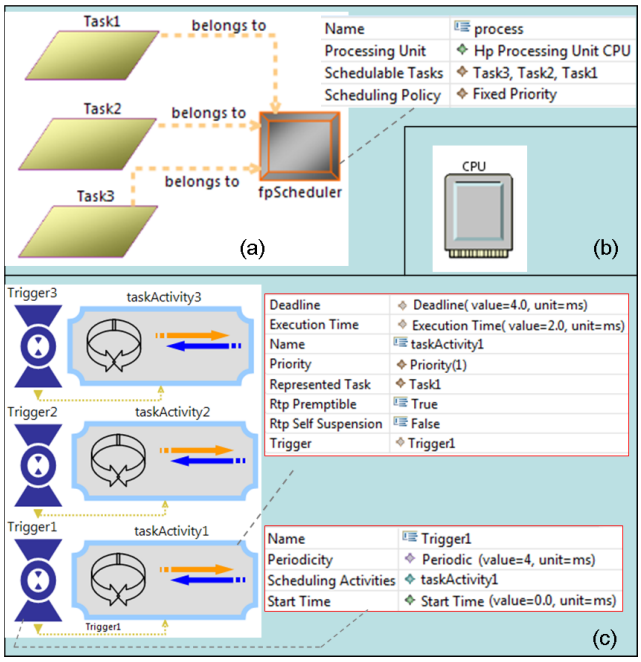


Figure 4. A simple real-time system that is designed using MoSaRT language: software model(a), hardware model(b) and behavioral model(c)

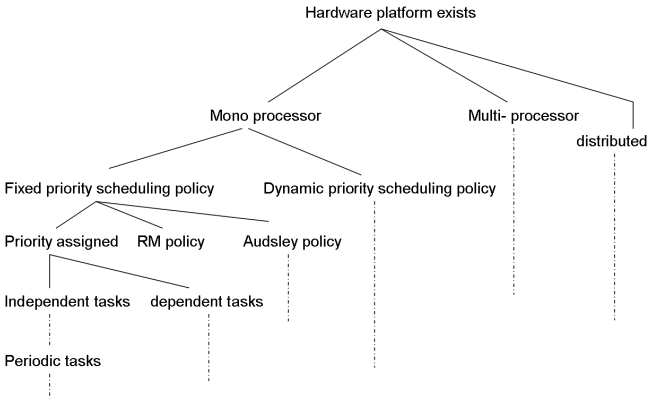


Figure 5. A decision tree of Scheduling analysis ability rules

rules are inter dependent and they are organized as a decision tree (see Figure 5). So, the verification of a rule requires the accuracy of other rules. For example, MoSaRT first checks if a hardware platform exists. If true, then it will verify if the hardware architecture is uniprocessor, else it will verify if the hardware architecture is multi-processor else it will deduce the hardware architecture is distributed. We note that a tree exploration gives the task model which corresponds to the design or else the nearest worst-case model. For instance, the design appearing in Figure 4 respects a set of rules which correspond to Liu and Layland task model [10] assumptions. Therefore, the simulation, the worst case response time calculation and the processor utilization calculation are the analysis tests corresponding to our design example. This

 ISBN: 978-1-61208-233-2

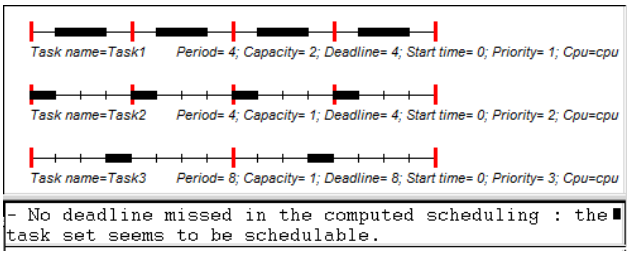


Figure 6. Result provided from Cheddar analysis tool

result does not mean that the system example is schedulable, but the system can be analyzed by a third-party tool providing the appropriate scheduling analysis tests. Figure 6 shows the simulation test applied by the real-time system example. This validation can be provided by Cheddar analysis tool.

V. Conclusion

We presented an approach to cope with design and analysis real-time system difficulty. MoSaRT is an implementation of this approach. It is a language which helps to design, to verify the system structure and to choose a schedulability test. It is based on Ecore and OCL. We presented two kinds of analysis that are based on a set of rules. In order to ease the understanding, the structural rules and scheduling analysis ability rules are presented through some examples. The transformation from MoSaRT to analysis tools is done manually. So, we are focusing on model transformation from MoSaRT to standardized design languages and then we will try to automatize the transformation process.

Acknowledgment

We thank Obeo that has provided us an academic version of Obeo-Designer product. We used this tool to implement our research works about the meta-modeling.

References

[1] OMG, “The uml profile for marte: Modeling and analysis of real-time and embedded systems,” www.omgmarte.org, [retrieved: Sept, 2012].

[2] SAE, “Architecture analysis and design language,” www.aadl.info, [retrieved: Sept, 2012].

[3] J. L. Medina Pasaje, M. González Harbour, and J. M. Drake, “Mast real-time view: A graphic uml tool for modeling object-oriented real-time systems,” in *RTSS 2001*, pp. 245–256.

[4] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, “Cheddar: a flexible real time scheduling framework,” in *SIGAda 2004*, pp. 1–8.

[5] A. K. Mok and D. Chen, “A multiframe model for real-time tasks,” *IEEE Transactions on Software Engineering*, vol. 23, pp. 635–645, 1996.

[6] J. C. Palencia and M. González Harbour, “Schedulability analysis for tasks with static and dynamic offsets,” in *RTSS 1998*, pp. 26–37.

[7] A. Rahni, E. Grolleau, and M. Richard, “An efficient response-time analysis for real-time transactions with fixed priority assignment,” *ISSE*, vol. 5, no. 3, pp. 197–209, 2009.

[8] F. Ridouard, P. Richard, and F. Cottet, “Negative results for scheduling independent hard real-time tasks with self-suspensions,” in *RTSS 2004*, pp. 47–56.

[9] M. Richard, P. Richard, E. Grolleau, and F. Cottet, “Contraintes de precedences et ordonnancement mono-processeur,” in *Real Time and Embedded Systems*, Teknea, Ed., 2002, pp. 121–138.

[10] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[11] S. Vestal, “Fixed-priority sensitivity analysis for linear compute time models.” *IEEE Trans. Software Eng.*, pp. 308–317, 1994.

[12] E. Bini, M. Di Natale, and G. Buttazzo, “Sensitivity analysis for fixed-priority real-time systems,” *Real-Time Syst.*, vol. 39, pp. 5–30, August 2008.

[13] N. Audsley and Y. Dd, “Optimal priority assignment and feasibility of static priority tasks with arbitrary start times,” 1991.

[14] R. Mishra, N. Rastogi, D. Zhu, D. Moss, and R. Melhem, “Energy aware scheduling for distributed real-time systems,” in *IPDPS 2003*, p. 21.

[15] M. de Roquemaurel, T. Polacsek, J.-F. Rolland, J.-P. Bodeveix, and M. Filali, “Assistance la conception de modles l’aide de contraintes,” in *AFADL 2010*, pp. 181–196.

[16] A. Plantec, F. Singhoff, P. Dissaux, and J. Legrand, “Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns,” in *ISOLA 2010*, pp. 4–17.

[17] F. Peres, P.-E. Hladik, and F. Vernadat, “Specification and verification of real-time systems using pola,” *IJCCBS*, pp. 332–351, 2011.

[18] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[19] OMG, “Meta object facility,” www.omg.org/spec/MOF/2.4.1/, [retrieved: Sept, 2012].

[20] Y. Ouhammou, E. Grolleau, M. Richard, and P. Richard, “Towards a simple meta-model for complex real-time and embedded systems,” in *MEDI 2011*, pp. 226–236.

[21] ——, “Model driven timing analysis for real-time systems,” in *ICESS 2012*, pp. 1458–1465.

[22] OMG, “Object constraint language,” www.omg.org/spec/OCL/2.0/, [retrieved: Sept, 2012].

 ISBN: 978-1-61208-233-2

Automated Structural Testing of Simulink/TargetLink Models via Search-Based Testing assisted by Prior-Search Static Analysis

Benjamin Wilmes
Berlin Institute of Technology
Daimler Center for Automotive IT Innovations (DCAITI)
Berlin, Germany
E-Mail: benjamin.wilmes@dcaiti.com

***Abstract*—Considering the advantages of early testing and the importance of an efficient quality assurance process, the automation of testing software models would be of great benefit, in particular to the automotive industry. Search-based testing has been applied to automate testing of Simulink models. Despite promising results however, the approach lacks efficiency. Working toward a robust tool, this paper presents three static-analysis-based techniques for assisting and improving search-based structural testing of TargetLink-compliant Simulink models. An interval analysis of model internal signals is used to identify unsatisfiable coverage goals and exclude them from the search. A further analysis determines which model inputs a coverage goal actually depends on in order to reduce the search space. We also propose a technique that sequences coverage-goal-related search processes in order to maximize collateral coverage and reduce test suite size. These additional techniques make the search-based approach applied to Simulink more efficient, as first experiments indicate.**

Keywords-Search-Based Testing; Static Analysis; Simulink.

I. INTRODUCTION

Today, in many application areas, the creation of embedded controller software relies on model-based design paradigms. Various industries, such as the automotive industry, use Matlab Simulink (SL) [1] as the standard tool to create and simulate dynamic models along with a code generator, for instance TargetLink (TL) [2], in order to automatically derive software code from such models.

As they are normally the first executable artifacts within software development processes, SL models play an important role in testing theory. Industrial testing practice however, usually focuses on higher-level development artifacts, like testing integrated software or systems as a whole. This discrepancy has both traditional and practical reasons. On one hand, current testing processes still require further adaptation to the model-based paradigm. On the other hand, companies are pressed for time in product development and must deal with an increasing demand for innovation. This can lead to a disregard for low-level tests and model tests in particular. Yet focusing too one-sided on tests of higher-level software or system artifacts poses the risk that faults may be found late in the process, which can lead to increased costs, that some faults can hardly be discovered on higher levels, or that certain functionality is not tested at all.

Thus, automating the testing of software models is highly desirable in industrial practice, particularly with regard to what is normally the most time-consuming testing activity: the selection of adequate test cases in the form of model input values (test data). Search-based testing is a dynamic approach to automating this task. It transforms the test data finding problem into an optimization problem and utilizes meta-heuristic search techniques like evolutionary algorithms to solve it. Search-based testing [3] has been studied widely in the past and has also been applied successfully for testing industrial-sized software systems [4]. Both structural (white-box) and functional (black-box) testing can be automated with the search-based approach.

Zhan and Clark [5], as well as Windisch [6], applied search-based testing to structural test data generation for SL models. The work of Windisch not only supports Stateflow (SF) diagrams (which are used fairly often in SL models), but also makes use of an advanced signal generation approach in order to generate realistic test data. While his approach has led to promising results in general, outperforming commercial tools, it lacks efficiency when applied to larger models. Furthermore, it shows difficulties targeting Boolean states and tackling complex dependencies within models [7].

The work presented in this paper is a first step toward overcoming some of these shortcomings by exploiting static model analysis techniques before the search process actually starts. Our scope is test data generation for TL-compliant Simulink models. We aim to improve both the efficiency and effectiveness of the approach by Windisch.

This paper is structured as follows: Section II introduces search-based testing and its application to structural testing of SL models. Section III presents our approach to supporting the search-based technique by integrating three additional analysis techniques. We present a signal range analysis (Section III-A), which captures range information of internal model signals, and in this way, allows partial detection of unreachable model states. We then propose a signal dependency analysis for the purpose of search space reduction (Section III-B). Our third contribution is a sequencing approach, which derives an order in which

 ISBN: 978-1-61208-233-2

coverage goals of a structural test are processed by the search (Section III-C). Insight into our implementation along with first experimental results is provided in Section IV, followed by our conclusions in Section V.

II. Background

A. Search-Based Structural Testing

Search-based testing [3] and its application to industrial cases has been extensively studied in the last decade, motivated by its general scalability, in contrast to purely static techniques like symbolic execution.

The general idea of search-based testing is pretty simple: a test data finding problem (which surely differs in its nature depending on the kind of testing) is transformed into an optimization problem by defining a cost function, called a fitness function. This function rates any test data generated by the deployed search algorithm - usually based on information gained from executing the test object with it. The rating must express in as much detail as possible, how far the test data is from being the desired test data. An iteratively working search algorithm uses these fitness ratings to distinguish good test data from bad, and based on this, generates new test data in each iterative cycle. This fully automated procedure continues until test data satisfying the search goal(s) has been found, that is, if a fitness rating has reached a certain threshold or until a predefined number of algorithm iterations have been performed. Various search algorithms have been used in the past. Due to their strength in dealing with diverse search spaces, evolutionary algorithms, like genetic algorithms, were often preferred [8].

Applied to functional testing, the search-based approach is generally utilized to search for violations of a requirement. In this case, a sophisticated fitness function needs to be designed manually when following the standard approach. However, when applied to structural test data generation, fitness functions can be derived completely automatically from the inner structure of the program to be tested.

Structural testing is commonly aimed at deriving test data based on the internal structural elements of the test object, e.g., creating a set of test data which executes all statements of a code function, or all paths in the corresponding control flow graph. Industrial standards like ISO 26262 even demand the consideration of coverage metrics when performing low-level tests. Search-based testing can automate this task for various coverage criteria (like branch or condition coverage) by treating each structural element requiring coverage as a separate search goal, called a coverage goal (CG). Each CG is accompanied by a specific fitness function. Wegener et al. [9] recommend composing the fitness function of the following two metrics: approach level (positive integer value) and branch distance (real value from 0 to 1). Given a test data's execution path in the control flow graph of the test object's code, the approach level describes the smallest number of branch nodes between the structural element to be covered and any covered path element. To create a more detailed and differing rating of generated test cases, the branch distance reflects how far the test object's execution has been from taking the opposite decision at the covered branch node, which is the closest to the structural element to be covered. This approach is suitable for structural testing of program code, like C or Java code.

B. Application to Dynamic Systems

As model-based development is now established in the automotive industry and practitioners have noticed opportunities to test earlier, Windisch [6] as well as Zhan and Clark [5] have transferred the idea of search-based structural testing from code to model level. For SL models, structural coverage criteria similar to the ones known from code testing exist and are commonly accepted in practice. Before addressing the challenges of applying search-based test data generation to SL models, we give a brief introduction to SL. SL is a graphical data-flow language for specifying the behavior of dynamic systems. Syntactically, a SL model consists of functional blocks and lines connecting them, while most of the blocks are equipped with one or more input ports as well as output ports. The semantics of such a model results from the composed functionalities of the involved block types, e.g., sum blocks, relational blocks or delay functions. In addition, event-driven or state-based functionalities can be realized within SL models using SF blocks. A SF block contains an editable Statechart-like automaton.

When applying search-based structural testing to SL models, two fundamental differences compared to its application on code level arise. First, SL models describe time and state dependent processes. Inputs and outputs of SL models, as well as block-connecting lines, are in fact signals. In order to enable reaching all system states, an execution with input sequences (signals) instead of single input values is required. Such complex test data can only be generated with common search algorithms by compressing the data structure, as done by Windisch [10]. His segment-based signal generation approach also considers the necessity for being able to specify the test data signals to be generated (e.g., amplitude bounds and signal nature like wave or impulse form). Second, the aforementioned fitness function approach cannot be fully adopted since SL models are data flow-oriented. There are no execution paths because the execution of a SL model involves the execution of every included block. Hence, a CG-related fitness function addresses only distances to the desired values of one or more model internal signals. For CGs in SF diagrams however, a bipartite fitness approach is possible [7].

Figure 1 visualizes the overall work flow of applying search-based test data generation to structural testing of SL models as described.

 ISBN: 978-1-61208-233-2

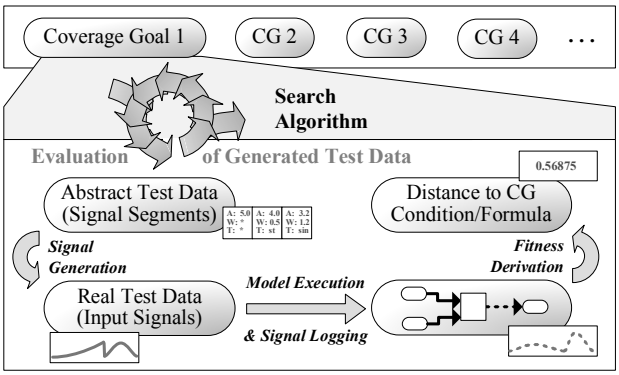


Figure 1. Automated search for test sequences, which fulfill coverage goals derived from the model under test

C. Deficiencies and Potential

Search-based structural testing has been applied successfully to real (proprietary) SL models originating from development projects at Daimler, e.g., a model of a windscreen wiper controller [7]. Compared to purely random test data generation of similar complexity, the search-based approach results in significantly higher model coverage. Even in comparison with a commercial tool, the search-based approach performs more effectively.

Despite promising results, the approach lacks efficiency. In general, the overall runtime of the search processes for achieving maximal model coverage increases with the size of the model under test. Similar experiences have been made with code-level search-based structural testing [11]. Since a single automotive SL model is often hundreds of blocks in size, and because a test data generation process of more than a couple of hours is undesirable, improving efficiency of the search-based approach is vital. Apart from the shortcomings of this approach that will be addressed by the contributions in the following sections, there are two other technical problems leading to a lack of efficiency. First, the structural test data generation is performed black-box-like, which means that the model is fed with input values on one end while some distances for calculating fitness are measured at some other point in the model. Any structural information between is not considered, thus the search might be blind to complicated dependencies in the model (cf. [12]). Second, when targeting a Boolean state in a model, a suitable fitness function is hard to find since a simple true or false rating inadequately leads a search [7]. Zhan and Clark suggest a technique called tracing and deducing [13], which mitigates this problem in certain cases, but fails in instances where the Boolean problem cannot be traced back in the model to a non-Boolean one. Further work will address these problems.

As a whole, we aim to improve the search-based approach for structural testing of SL models so that it performs acceptably and reliably in industrial development environments. To this end, we turn our attention to testing of TL-compliant SL models since the code generator TL is widely used in industrial practice. TL extends SL by offering additional block types, but also makes restrictions on the usage of certain SL constructs like block types. Nevertheless, it is possible to adapt our ideas to pure SL usage.

III. Prior-Search Static Analysis

We distinguish between techniques which support search-based structural testing (a) before the CG-related search processes, (b) between the different search processes, (c) during each search process, and (d) after the search processes are done. In the following sections, three techniques belonging to category (a) are presented. Apart from making use of an input specification and choice of coverage criteria provided by the user, all three techniques are fully automatic.

A. Signal Interval Analysis

In structural testing practice, achieving 100% coverage is often not possible. One reason lies in the semantic constructions precluding certain states or signal values. It might also be that a tester specified the test data to be generated in such a way that it prevents certain CGs from being satisfiable. Also, SL models might be designed variably, e.g., contain a constant block with a variable value. When such variability is bound during execution, e.g., via configuration file, certain model states may be unreachable.

CGs referring to unreachable states worsen the overall runtime and undermine the efficiency of search-based structural test data generation since time-consuming search processes are carried out without any hope of finding desired test data. Therefore, we propose two techniques contributing to automatic identification of unreachable CGs. The first one is an interval analysis, which determines the range within which the values of every internal model signal are. If a signal range is in conflict with the range or value required by a CG, this CG is unsatisfiable. We use interval analysis since other approaches to detect infeasibility, such as constraint solving or theorem proving [14], are currently not scalable enough for the complex equations constituted by industrial-sized SL models. The second technique is an analysis of dependencies between CGs. Since this technique is mainly used for another purpose, it is presented in Section III-C.

The code generator TL, as well as the latest version of SL, are capable of analyzing signal ranges in order to perform code optimizations and improve scaling or data type selection, respectively. While those range analysis features are limited (e.g., determining ranges of signals that are involved in loops is not possible without user interaction) our signal interval analysis (SIA) makes use of an input signal specification in order to overcome such limitations and derive more precise ranges.

As mentioned in Section II-B, a tester who uses the search-based approach for testing SL models, as outlined by

 ISBN: 978-1-61208-233-2

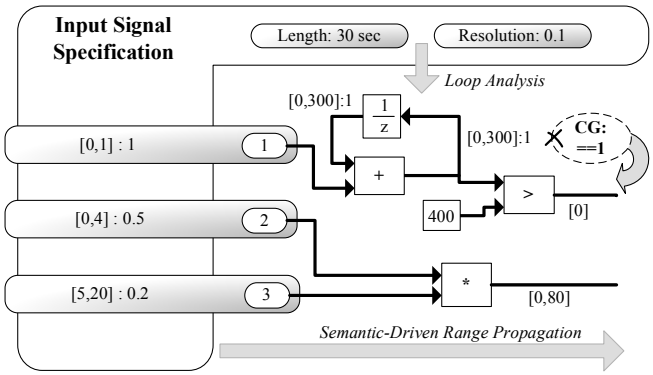


Figure 2. Example of how determining the ranges of a model's internal signals based on input specification and block semantics works

Windisch, is asked to specify the test data to be generated first. This involves establishing (a) the range boundaries and step size of each model input as well as defining (b) a common length (in seconds) and sample rate for all input signals. SIA starts with information (a) for a model's input signals and propagates the corresponding signal ranges of the form $[x, y] : q$, where q (optional) is the step size, through the whole model. Following Wang et al. [15] we also use interval sets instead of a single interval per signal in order to derive more accurate range information. Each propagation step is based on the semantics of the block connecting the signals. In this context, we derived interval semantics for each block type of TL-compliant SL models using basic concepts of interval arithmetic [16].

Figure 2 graphically depicts this procedure with the aid of a simple example. Note that the model contains a loop, initiated by a delay block with an initial value of 0. The standard propagation procedure would be unable to continue here since ranges are not available for all incoming signals of the sum block. A simple, yet imprecise solution is to set the range of the sum block's outgoing signal to the minimum and maximum values of the signal's data type. A more precise solution however, is to use the information (b) of the signal specification in order to run a loop analysis. From length and sample rate, the number of loop iterations is derivable. Starting with the initial value of the delay block a static analysis of the loop iterations is performed, resulting in time-related range information. In order to keep the final results clean and minimal, each signal's ranges as well as the time phases of ranges are combined, if possible, in each iteration of the loop analysis.

Using range propagation and loop analysis in combination, SIA is capable of determining the ranges of all signals contained in the model under test. In cases of blocks with unknown semantics or unsupported blocks, the minimum and maximum values of the outgoing signal's data type are used. Finally, the results of SIA are used to assess whether each CG's associated formula is unsatisfiable - see the exemplary CG in Figure 2. In addition to unsatisfiable CGs, SIA can also identify Boolean signals or discrete signals with only a few possible different values. As described in Section II-C, CGs related to such signals could be problematic for the search-based approach.

B. Signal Dependency Analysis

By default, the search algorithm generates test data for all of the model's inputs when targeting a CG. However, there are usually CGs whose satisfaction is, in fact, independent of the stimulation of certain model inputs. By not taking this into account, the search space is unnecessarily large, which makes it more difficult for the search to find desired test data. To raise efficiency, we include a signal dependency analysis (SDA) to identify which model inputs each CG actually depends on. McMinn et al. [17] investigated a related approach, however, on code level. SDA is closely related to a slicing approach for SL models developed in parallel to our work [18].

At code level, such analysis is usually done by capturing the control dependence in a graph. SL models though, as pointed out previously, are dominated by data dependencies. We therefore analyze the dependency of CGs on input signals by creating a signal dependency graph (a) based on the syntax of the model and (b) refined according to the semantics of blocks which have multiple outgoing signals. Focusing purely on syntax, the following principle leads to a graph describing which signal b the value of a model internal signal a depends on: Signal a is dominated by a signal b if signal a is the outcome of a model block which has signal b incoming. Some blocks with multiple outgoing signals however, do not use every incoming signal in order to calculate the value of a certain outgoing signal. In such cases, the signal dependency graph is refined by removing over-approximated dependencies.

In order to determine which model inputs a certain CG depends on, the signal or signals, which the CG expression refers to, are selected in the dependency graph first. By traversing the graph up to the input signals, the set of relevant model inputs is collected. Within the subsequent search process for this CG, signals are generated only for the relevant inputs - all other inputs receive a standard (e.g., zero valued) signal when being executed.

C. Coverage Goal Sequencing

No matter if structural testing is performed in addition to functional (black-box) testing or purely as white-box testing, it is usually a set of CGs that constitutes the test objective. Remember, that for each CG a separate search needs to be run. In Windisch's approach, those search processes are executed in random order. Hence, correlations between CGs are ignored. Given CGs with the expressions $s{<}90$, $s{<}80$ and $s{<}70$, for example, it is most likely more efficient to aim for reaching the goal $s{<}70$ first because it satisfies all

 ISBN: 978-1-61208-233-2

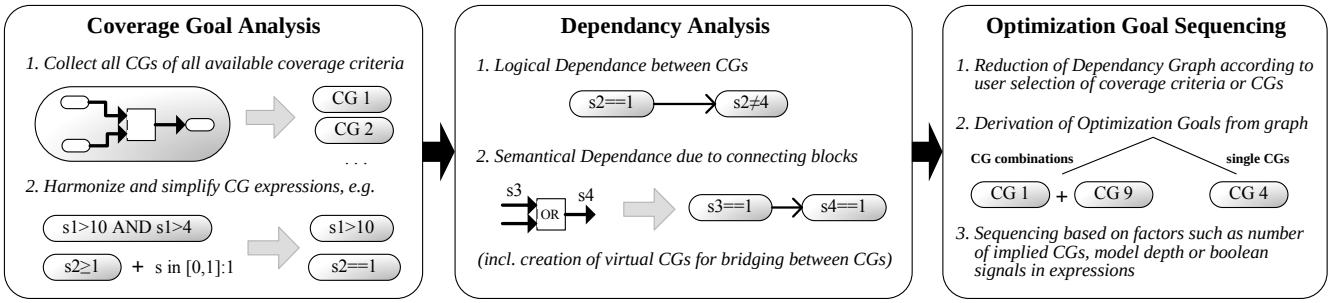


Figure 3. Main process steps to create an efficient order for a set of coverage goals, which are processed separately by a search

other CGs at the same time. As this example indicates, the execution order of the CG-related search processes affects the efficiency of the whole structural test.

Other researchers in the search-based testing community have noticed this shortcoming as well. Fraser and Arcuri [19] advise focusing on the generation of whole test suites rather than targeting single CGs. They recommend optimizing multiple test suites instead of multiple test data, and also suggest rewarding smaller test suites with a better fitness in case two or more test suites achieve the same coverage. Harman et al. [20], in contrast, suggest a multi-objective search in which each CG is still targeted individually but the number of collateral (accidentally covered) goals is included as a secondary objective. Though facing a similar problem, our approach differs. We keep the focus on single CGs since, considering the complexity of the optimization problems constituted by industrial SL models, they are often difficult to reach and we do not want to impede the search by burdening it with additional goals or mixed fitness values. Instead, we propose a coverage goal sequencing approach that creates a reasonable order in which the various CGs are pursued. Li et al. worked out a related approach [21], however it is outside of the search-based and SL context. Ultimately, by maximizing collateral coverage, our approach attempts to minimize the number of CGs that need to be pursued. Not only is this expected to improve overall efficiency, but the resulting test suite should also be smaller.

In this paper, we can only give a brief introduction to this technically complex solution, as summarized in Figure 3. First of all, the model under test is analyzed and CGs are derived for all SL/SF-relevant coverage criteria (see [7]). In preparation to analyzing dependencies between CGs, we apply several harmonization and simplification steps to the CG expressions. Note that results of SIA (Section III-A) are used for this task as well, e.g., an expression $s \geq 1$ would be transformed to $s=1$ if s is a Boolean signal. Next, possible dependencies between CG expressions are analyzed, resulting in a set of dependency graphs. A set, since only certain dependencies are considered in order to limit complexity. The following CG relations are registered: implication, equivalence, NAND, and XOR. Dependencies of the expressions are analyzed both from a logical and a semantical point of view. Semantical means that for each two CGs relating to incoming or outgoing signals of the same model block, a block-specific analysis checks if a relation between the CGs exists. In certain cases, the block-specific analysis adds virtual CGs as a bridge to other CGs in order to detect further dependencies. Along the way, based on the captured dependencies, further CGs might be detected as unsatisfiable (see Section III-A).

Considering the user's selection of coverage criteria or single CGs, the graphs are minimized accordingly. Amongst others, non-selected CGs implying selected CGs are kept. From the graphs the final optimization goals are derived in a two-fold way: Besides keeping each selected CG as a single optimization goal, certain combinations of CGs are derived as well - since a graph can contain conjunctions. In this way, optimization goals with high collateral coverage are added. Finally, the optimization goals are sequenced according to several metrics, primarily by the number of (so far unsatisfied) implied CGs, but also by their depth in the model and the amount of Boolean signals involved in the expressions - since such goals should be avoided given the fitness function construction problem (see Section II-C). Note that the pursuing order of optimization goals is updated after each search process ends, since an optimization goal's number of unsatisfied implied CGs might have changed.

IV. Implementation And First Experiments

The presented analysis techniques have been implemented in the course of developing our tool TASMO [22], which is implemented in Java and closely integrated with Matlab. TASMO extracts model related information from Matlab and applies transformation and reduction steps to an internal representation of the model under test in order to focus on the relevant parts for structural test data generation. TASMO can also visualize the results of the presented analysis techniques. We investigated the effect of these techniques to structural test data generation for industrial SL/TL models. Case studies applying the whole procedure, including running the search procedure, are part of ongoing work. We present first experimental results of applying signal

 ISBN: 978-1-61208-233-2

interval analysis and signal dependency analysis to two SL models, one mid-sized (model A, 147 blocks, 323 CGs) and one large-sized (model B, 1047 blocks, 736 CGs), recently developed at Daimler in the scope of an electric vehicle's propulsion strategy. Based on the input specifications, SIA identified 17 (A) and 39 (B) infeasible CGs for which the standard search-based approach would otherwise perform extensive search processes. SDA detected that for each CG, on average, only about 3.1 out of 8 (A) and 17.3 out of 32 (B) model inputs would need to be stimulated in order to match the CG formulas. This shows how this technique can reduce the search space distinctly - without excluding CG-relevant search space areas. The runtime of the additional analysis techniques was only a matter of seconds.

V. Conclusion and Future Work

This paper introduces an approach to improving the performance of search-based testing when applied to structural testing of SL models. Three static techniques extend the standard search-based approach by analyzing the model under test before the search processes for each CG are run. Unsatisfiable CGs are partially identified and excluded from the search. The search space is reduced in such a way that the search focuses solely on relevant model inputs. The separate search processes for each CG are sequenced in order to maximize collateral coverage, minimize test suite size, and shorten the overall search runtime.

First experiments backed up our expectations and extensive case studies will follow in the course of our proceeding tool development. We aim to develop a prototype tool that is applicable in industry. Further work is required to extend the presented techniques with full Stateflow support. Targeting the discussed shortcomings of the search-based approach, our next main step is to work on a hybridization of the search-based algorithm with supporting (static) techniques.

References

[1] The Mathworks, "Matlab Simulink," Last access: 2012-09-16. [Online]. Available: http://www.mathworks.com

[2] dSpace, "Targetlink," Last access: 2012-09-16. [Online]. Available: http://www.dspace.com

[3] P. McMinn, "Search-based software testing: Past, present and future," in *Softw. Testing, Verif. and Valid. Workshops (ICSTW)*, 2011.

[4] B. Wilmes, A. Windisch, and F. Lindlar, "Suchbasierter Test für den industriellen Einsatz," in *4. Symp. Test. im Sys.- und Softw. Life-Cycle*, 2011.

[5] Y. Zhan and J. A. Clark, "A search-based framework for automatic testing of MATLAB/Simulink models," *J. Syst. Softw.*, vol. 81, no. 2, pp. 262–285, Feb. 2008.

[6] A. Windisch, "Search-based testing of complex simulink models containing stateflow diagrams," in *Proc. of the 1st Int. Workshop on Search-Based Soft. Testing*, 2008.

[7] A. Windisch, "Suchbasierter Strukturtest für Simulink Modelle," Ph.D. dissertation, Berlin Institute of Technology, 2011.

[8] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Trans. on Softw. Eng.*, vol. 36, pp. 226–247, 2010.

[9] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Inform. and Softw. Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[10] A. Windisch and N. Al Moubayed, "Signal generation for search-based testing of continuous systems," in *Softw. Testing, Verif. and Valid. Workshops (ICSTW)*, 2009.

[11] T. Vos, A. Baars, F. Lindlar, P. Kruse, A. Windisch, and J. Wegener, "Industrial scaled automated structural testing with the evolutionary testing tool," in *Proc. of the 3rd Int. Conf. on Softw. Testing, Verif. and Valid.*, 2010.

[12] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to search based test data generation," in *Proc. of the 2006 Int. Symp. on Softw. Testing and Analysis (ISSTA)*, 2006, pp. 13–24.

[13] Y. Zhan and J. A. Clark, "The state problem for test generation in simulink," in *Proc. of the 8th Annual Conf. on Genetic and Evolutionary Computation*, 2006, pp. 1941–1948.

[14] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," in *Proc. of the Int. Symp. on Softw. Testing and Analysis*, 1994, pp. 80–94.

[15] Y. Wang, Y. Gong, J. Chen, Q. Xiao, and Z. Yang, "An application of interval analysis in software static analysis," in *IEEE/IFIP Int. Conf. on Embedded and Ubiquitous Computing*, vol. 2, 2008, pp. 367 –372.

[16] R. E. Moore, *Interval Analysis*. Prentice-Hall, 1966.

[17] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener, "Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation," *IEEE Trans. on Softw. Eng.*, vol. 38, pp. 453–477, 2012.

[18] R. Reicherdt and S. Glesner, "Slicing Matlab Simulink models," in *34th Int. Conf. on Softw. Eng.*, 2012, pp. 551 –561.

[19] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *11th Int. Conf. on Quality Softw.*, 2011.

[20] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Softw. Testing, Verif. and Valid. Workshops (ICSTW)*, 2010, pp. 182–191.

[21] J. J. Li, D. Weiss, and H. Yee, "Code-coverage guided prioritized test generation," *Inf. Softw. Technol.*, vol. 48, no. 12, pp. 1187–1198, 2006.

[22] B. Wilmes, "Toward a tool for search-based testing of Simulink/TargetLink models," in *4th Symp. on Search Based Softw. Eng. (Fast Abstracts)*, 2012.

 ISBN: 978-1-61208-233-2

Fault Detection Capabilities of an Enhanced Timing and Control Flow Checker for Hard Real-Time Systems

Julian Wolf, Bernhard Fechner and Theo Ungerer
University of Augsburg, Germany
Emails: {wolf, fechner, ungerer}@informatik.uni-augsburg.de

***Abstract*—Dependability and robustness are essential requirements of embedded systems. It is necessary to develop and integrate mechanisms for a reliable fault detection. Regarding the context of hard real-time computing, such a mechanism should also focus on the correct timing behavior. In this paper, we present results of the fault detection capabilities, i.e., the fault coverage and detection latencies, of a novel timing and control flow checker designed for hard real-time systems. An experimental evaluation shows that more than 65 % of injected faults uncaught by processor exceptions can be detected by our technique – at an average detection latency of only 22.1 processor cycles. Errors leading to endless loops can even be reduced by more than 90 %, while the check mechanism causes only very low overhead concerning additional memory usage (15.0 % on average) and execution time (12.2 % on average).**

Keywords-Control flow checking; timing correctness; reliability; embedded processors; hard real-time computing

I. INTRODUCTION

Most deployed systems in safety-critical areas, like the automotive and aerospace domains, are hard real-time computing systems. They must provide analyzable timing behavior, because missing a deadline potentially causes catastrophic consequences. The primary goal is not to optimize the average performance, but to provide analyzability and to determine timing guarantees [20]. If an embedded system is intended for safety-critical applications, designers must also guarantee that soft errors, e.g., caused by transient faults, have negligible impact on the execution behavior. It is necessary to integrate reliable error detection mechanisms with low detection latency enabling an immediate reaction to any misbehavior and possibly the execution of a fall-back solution within the required deadlines.

In this context, we focus on errors occurring in the control path of an embedded hard real-time processor, i.e., errors causing timing or logical divergence from the proper control flow. Fault injection studies show that up to 77 % [19] of errors occurring in a computer system are control flow errors. Regarding system errors caused by transient, non-reproducible faults, an on-line error detection mechanism is the only feasible solution to detect such errors.

In [23] and [24], Wolf et al. provide a detailed description of a novel timing and control flow check mechanism. The approach extends fine-grained on-line timing checks for hard real-time systems by a lightweight control flow monitoring technique. The instrumentation of application code at compile-time is combined with a small hardware check unit connected to the core verifying the correctness at run-time.

In this paper, we enhance this approach by additional checks of a lower timing bound. Moreover, we particularly focus on the fault detection capabilities of the check mechanism. A fault injection study based on automotive benchmarks provides additional results showing the main benefits of the approach, i.e., a wide coverage of possible soft errors resulting in a reduction of critical system failures, very low fault detection latencies, and low memory and execution time overhead.

This paper is organized as follows: Section II summarizes related work in the field of on-line checking techniques. Our proposed method for temporal and logical control flow monitoring is presented in Section III. Subsequently, Section IV shows details on implementation issues. The results of fault injection experiments are presented in Section V. Finally, Section VI concludes this paper and gives an outlook to future work.

II. RELATED WORK

Several methods for control flow checking – neglecting timing correctness – have been proposed during the last decades, implemented in hardware or software. Accordingly, these approaches either introduce additional hardware, like a watchdog processor [12] performing reliability checks during run-time [11], [15], [19], [22], or they add supplementary code on software-level to perform monitoring operations [1], [8], [14], [18]. However, both alternatives have benefits and drawbacks as well: While hardware-based approaches usually provoke high complexity for the integration into a system, their advantage is a good average performance due to less overhead concerning memory usage and execution time. Moreover, most of these techniques do not require changes in the executed application. Software-based approaches on the other side are easy to integrate, but cause significant overhead. Also, it is needed to add redundant information to the application source code, given that it is available. A solution for this dilemma can be the usage of a *hybrid* detection technique [4], [17], combining benefits of both hardware- and software-based approaches.

 ISBN: 978-1-61208-233-2

Figure 1. Temporal instrumentation of basic blocks

On the other hand, Paolieri and Mariani [16] introduce a special hardware unit to support timing correctness at system level. The developed *timing-aware coverage monitor unit* is CPU-independent, but requires timing footprints of the running task. However, this approach focuses mainly on timing errors caused by a multi-threaded usage of commonly used resources, but not on transient faults. The intention of this technique is only to guarantee timing correctness while completely neglecting logical aberrations from the proper control flow.

III. Detection Mechanism

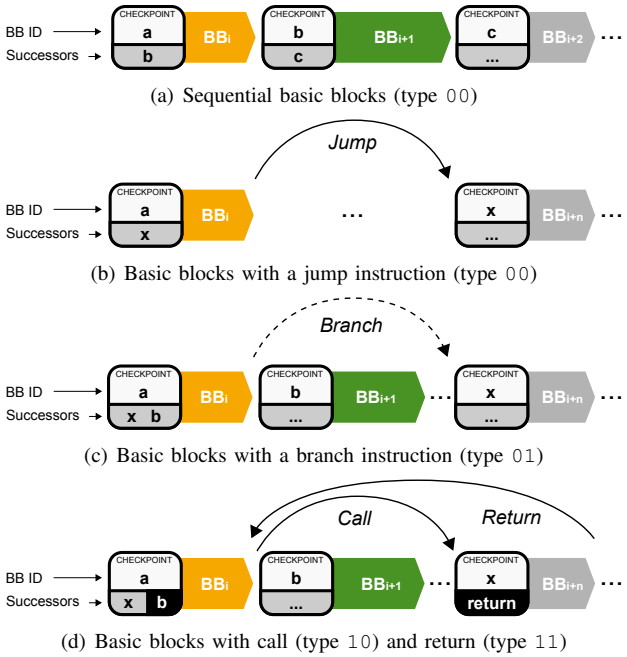
Our hybrid timing and control flow checking mechanism consists of two phases: In an *off-line* phase, the safety-critical application is split into basic blocks (BB), i.e., sequences of instructions in which the execution always begins at the first and terminates at the last instruction. These blocks are analyzed and hardened with instrumented checkpoints in the object code. In an *on-line* phase, a connected hardware check unit reacts to the inserted checkpoints during execution. If the program flow does not correspond to the instrumented information, an error is signaled.

We separate the description of our technique into two parts: Firstly, we explain the instrumentation and checking mechanism only for timing errors occurring in the control flow. Secondly, the additional part focusing on the detection of logical control flow errors is presented.

A. Temporal Control Flow Monitoring

After splitting the code into basic blocks, we add checkpoints at the beginning of each block containing information about its timing behaviour. In detail, this timing information consists of a lower bound symbolizing the *Best-Case Execution Time* (BCET) estimate and an upper bound, the *Worst-Case Execution Time* (WCET) estimate (see Fig. 1).

In the on-line phase, a specific hardware check unit transfers the timing values to defined registers, as soon as a checkpoint is reached. The register value symbolizing the WCET is decremented at each following processor cycle. When the next checkpoint is reached, the register is updated by the next WCET value. Therefore, we can assume a timing error, if the register is below zero. In this case, a basic block required more cycles than the WCET analysis had computed off-line. For checking the minimum execution time, a counter value is set to zero at the beginning of each basic block and is incremented at each following processor cycle. If the counter value is lower than the instrumented BCET bound when reaching the following checkpoint, a timing error occurred.



(a) Sequential basic blocks (type 00)

(b) Basic blocks with a jump instruction (type 00)

(c) Basic blocks with a branch instruction (type 01)

(d) Basic blocks with call (type 10) and return (type 11)

Figure 2. Logical instrumentation of basic blocks

B. Logical Control Flow Monitoring

If an application is split into basic blocks, their sequence during execution is analyzable. We annotate each basic block with a unique identifier (ID), which is added to the checkpoint containing the timing bounds. In order to enable a fast and easy check during run-time, we develop a technique to explicitly signalize successors: Along with each ID, we store the pre-calculated successor ID or two possible successor IDs of this basic block. So, the hardware checker compares in the on-line phase, if an actually executed basic block is an allowed successor. To give a better understanding, we regard each variation of the control flow, according to Fig. 2:

- In the *sequential* case (see Fig. 2(a)), we add to each checkpoint the ID of the basic block itself and the ID of its follower.
- In case of an unconditional (direct) *jump* instruction at the end of basic block BB_i in Fig. 2(b), the signalized successor of BB_i has to be updated accordingly.
- If a basic block ends with a *branch* or *loop* instruction, we cannot distinguish off-line which path will be taken during execution. So, basic block BB_i in Fig. 2(c) contains the IDs of both basic blocks BB_{i+1} and BB_{i+n} in a list of successors.
- Fig. 2(d) shows the instrumentation of *calls* and *returns*. In this example, BB_{i+n} is a function, which is called from BB_i. First, we add the ID of BB_{i+n} as the only allowed successor. Moreover, we append the basic block, which should be executed after the function's return (in this example BB_{i+1}). Within the function, it is sufficient just to signal the return. This instrumenta-

 ISBN: 978-1-61208-233-2

tion mechanism also works properly for nested function calls, if we introduce a stack memory to save multiple return IDs.

Since coding guidelines [5] for safety-critical hard real-time systems forbid the usage of indirect jumps and recursion due to problems concerning analyzability, we can neglect these issues in our context.

The hardware check unit, which becomes active as soon as a checkpoint is detected during run-time, is enhanced to interpret the instrumented values. The check unit has to verify, if the current ID corresponds to the signalized successor(s). Furthermore, it must save the current values for the checking progress when the application reaches the next checkpoint. We have to provide memory for storing at most two successors and a stack memory for function calls, which is dependent on the degree of function nesting.

IV. Implementation Issues

To evaluate strengths and weaknesses, we provide a tool for code instrumentation, which is integrated into the compilation process to enhance the assembly code of an application by checkpoints. This output is assembled and linked in order to get a binary that can be executed on a processor extended by a hardware check unit. The instrumentation could also be implemented on binary level. This might be useful, if application sources are not available, which can be neglected in our case.

A. Evaluation Platform

As a baseline for the execution we use the real-time capable multithreaded two-way superscalar CarCore processor [13]. The architecture of the CarCore is binary compatible to the Infineon TriCore [10], which is a commonly used microcontroller in safety-critical applications of the automotive industry. Up to two instructions per cycle can be assigned to its two pipelines (an *address* and a *data* pipeline) consisting of a *decode*, *execution* and *write back* stage. Both pipelines share the stages *instruction fetch* and *schedule* in the front part of the processor. The processor works in-order; instructions in both pipelines can be executed in parallel if an address instruction directly follows a data instruction.

Our simulations are executed on a cycle-accurate CarCore SystemC model, which exactly implements the timing behavior of the processor. This enables a measurement and comparison of realistic execution times of different applications.

B. Integration of Checkpoints

To handle the described methodology for timing and control flow checking, we need to enhance the application code by *BCET* and *WCET* estimates of a basic block, its *ID* along with two potential *Successor IDs*, and a field *Type* signalizing the required compare operation to the hardware check unit (see Fig. 2 for type values). However, we can avoid an explicit prediction of a second ID by a constraint on the assignment of basic block IDs: Each succeeding basic block in the assembler code should get an ID incremented by one (compared to its predecessor). By this, the second possible successor in a branch is always the ID of the block itself, incremented by one. Equally, we can implicitly define the return ID in case of a function call. The ID of the basic block of the return target is always the calling basic block's ID incremented by one.

Type	ID	Succ. ID	BCET	WCET
2 Bit	9 Bit	9 Bit	6 Bit	6 Bit

Figure 3. A 32 bit checkpoint

The timing bounds are computed with an analysis tool, which accumulates execution times of single instructions. The instrumentation tool can be enhanced by connecting a WCET tool providing less overestimation like the static WCET tool OTAWA [3], which also works on the baseline of basic blocks.

Focusing on low overhead, we choose an overall checkpoint bit width of 32 bit. This allows writing a checkpoint value to a 32 bit register of the CarCore processor, which has to be read by the hardware check unit. The bit mask displayed in Fig. 3 shows our implementation of a checkpoint: We need 2 bit for the declaration of the checkpoint type and 9 bit both for encoding the ID and the successor ID. The remaining 12 bits are used for the integration of the BCET and WCET values. This configuration allows a representation of 512 unique basic block IDs, which is sufficient regarding our evaluations.

To enable the check mechanism during run-time, it is necessary to add processor instructions, which trigger the check mechanism during execution. The CarCore processor provides special registers, called *Core Special Function Registers* (*CSFRs*) for hardware extensions. So, we implement the check unit to be triggered, as soon as an `MTCR` (move to core register) instruction on a specific checkpoint register is executed. For each checkpoint, we need three instructions: first, we write the checkpoint value into a data register (requires two instructions), then we call `MTCR` (one instruction) to copy the value to the special register. Since `MTCR` is an address instruction immediately following after a data instruction, the CarCore can execute two of these instructions in parallel, which minimizes execution time overhead.

C. Hardware Check Unit

To perform timing and control flow checks at run-time, we connect our hardware check unit directly to the processor pipeline. It needs two input signals: the processor clock and the decoded `MTCR` instructions including the checkpoint values. To estimate the hardware overhead of the integrated check unit, we transformed the SystemC code of the checker

 ISBN: 978-1-61208-233-2

to Very High Speed Integrated Circuit Hardware Description Language (VHDL) [21] and performed a synthesis for an Altera Stratix II Field Programmable Gate Array (FPGA) [2]. A critical point is the stack, which is needed for the call and return mechanism. Its size depends on the call depth of the program. If we store the stack in an on-chip RAM, which is cheaper than logic registers, the check unit requires only 163 Adaptive Look-Up Tables (ALUTs) (0.5 % compared to the overall CarCore processor) and 102 (1.0 %) logic registers, independent of the call depth. However, in this case we have an additional memory overhead of $(call\_depth * 9 \text{ Bit})/8$ Byte for the stack.

Currently, interrupts are neglected in our implementation. However, it is possible to extend the stack of the hardware check unit in order to support a kind of context change in case of an interrupt. But this will be part of our future work.

V. Experimental Evaluations

In this section, we focus on a detailed analysis of the implemented timing and control flow checking technique. In detail, we evaluate the detection coverage and latency using simulations with fault injections and we measure the overhead caused by the proposed mechanism. All evaluations are performed on the SystemC model of the CarCore processor, which was enhanced by the presented hardware check unit. Moreover, we integrated an extension enabling a systematic fault injection.

A. Benchmark Programs

We use different applications of the Embedded Microprocessor Benchmark Consortium (EEMBC) AutoBench 1.1 benchmark suite [7]. These programs are implemented in standard C and represent typical properties and requirements of automotive software for embedded systems. For the compilation, we use the *HighTec GNU C/C++ Compiler* for Infineon's TriCore (optimization level O2 enabled) [9].

B. Fault Model

Around 80 % - 90 % of hardware errors are induced by transient faults [6]. Therefore, in this context, we focus on transient faults in the form of Single Event Upsets (SEUs) during operation. These SEUs, usually appearing as bit flips, are presumed to occur in the instruction memory, since the consequences are very heterogeneous and challenging for a successful detection in such cases. As multiple bit faults at a time are extremely seldom, we assume only one single occurrence per program execution. For the fault injection studies, we modified the fetch stage of our simulated processor pipeline to inject bitflips. As the memory footprint of the EEMBC benchmarks is quite low, we can iteratively run simulations with a systematic injection of all potential bit flips in the instruction memory. Allover, we performed 143,673 simulation runs, each containing one bit flip.

C. Fault Coverage

As a first result of our evaluation studies, we observed that 67.0 % of injected faults cause an error, i.e., a deviation from the correct program functionality. In 33.0 % of all simulation runs, the injected faults showed no effects. This mainly results from the following causes:

- Since, according to our fault model, bit flips are injected in the fetch stage of the processor (always fetching 64 bit, i.e., up to four instructions at once), the faulty instructions are often not executed, e.g., in case of previously executed control flow instructions.
- The TriCore instruction set contains several unused bits in opcodes, where a bitflip will cause no erroneous behaviour, too.
- If a bitflip is injected inside an instruction representing a checkpoint, a shortening of the instrumented BCET value / an elongation of the WCET value will neither be detected nor lead to an error.

If we focus only on injected faults leading to errors, we can see that 28.4 % of these simulations abort due to an exception by the processor (19.5 % illegal opcode, 8.9 % wrong memory access). We can also neglect this part in our following considerations, since an additional error detection is not necessary in these cases.

In the first line (A) of Fig. 4 we can finally see the detection coverage of our proposed check mechanism – regarding errors, which are not caught by a processor exception. The results show that a total of 65.3 % can be detected: 41.1 % by logical control flow checks because of a wrong order of IDs, 19.4 % by timing checks due to an exceeding of the instrumented WCET estimates and 4.8 % by timing checks due to a deviation from the BCET values. On the other hand, 33.1 % of simulation runs terminate with wrong results. These are mostly pure data errors, which cannot be covered by our mechanism. Finally, we see 1.6 % of undetected endless loops; these loops comprise multiple basic blocks, since loops within one single basic block could be easily detected by our temporal check mechanism.

To compare the results to a system without our check mechanism, we also conduct a fault injection study using the EEMBC benchmarks without modifications. As these applications use less instruction memory due to the missing instrumentation, a less number of different bit flips can be injected (a total of 83,782 instead of 143,673). However, we can see a similar percentage of 38.7 % injected faults without any effects on the program behaviour. Regarding the remaining 61.3 % of simulation runs, there is a somewhat higher rate of 39.6 % of detections by processor exceptions. This increase is caused by the low detection latency of our check mechanism: Since several errors are detected very early, these errors can no longer raise a processor exception several cycles later. Focusing on errors, which are not caught by exceptions (see Fig. 4 (B)), there would be 83.2 % of

 ISBN: 978-1-61208-233-2

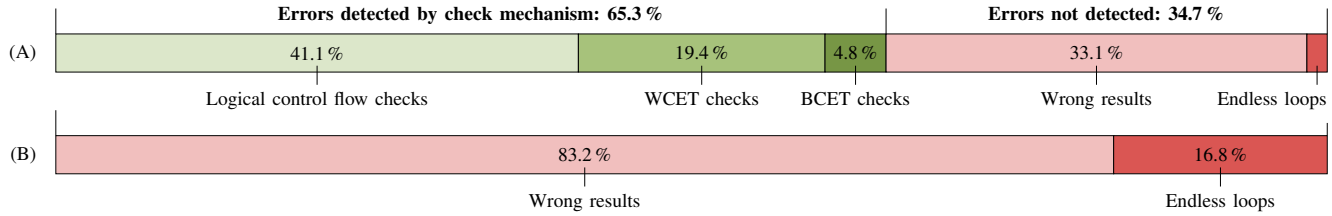


Figure 4. Behaviour of erroneous executions with (A) and without (B) integration of the proposed detection technique (results based on injected faults leading to errors, which are not caught by processor exceptions)

simulation runs terminating with wrong results and 16.8 % causing an endless loop.

Finally, we can conclude that the integration of our check mechanism reduces the number of errors leading to wrong results by more than 60 % (from 83.2 % to 33.1 %). The rate of errors leading to undetected endless loops even decreases by more than 90 % (from 16.8 % to 1.6 %).

D. Detection Latency

Beside the coverage, we evaluate the latency of our detection mechanism, i.e., the amount of processor cycles between fault injection and error detection. Focusing on latencies lower than 100 cycles (which make around 95 % of all executions), we receive a distribution shown in Fig. 5. Overall, the simulations show an average detection latency of 22.1 processor cycles. Values resulting from logical checks are generally lower (16.5 cycles on average) than those resulting from timing checks (25.4 cycles on average by BCET checks, 34.1 cycles by WCET checks). This difference is easy to explain: While an error is detected by logical checks directly after reaching a checkpoint with a wrong ID, an exceeding of the allowed execution time can only be detected when the estimated WCET bound of a basic block was overrun. The fact that several detections in Fig. 5 have a latency of around 70 cycles is a consequence of the call and return handling of the CarCore, which takes a high execution time compared to other architectures.

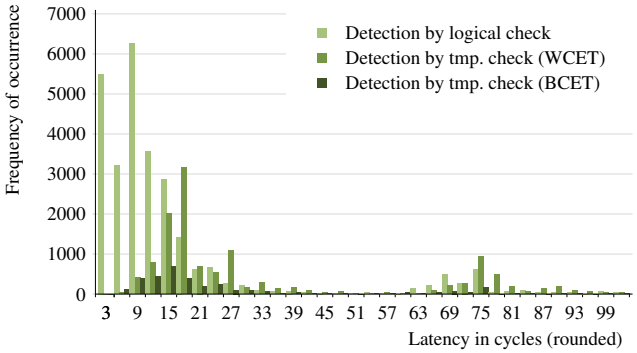


Figure 5. Distribution of detection latencies

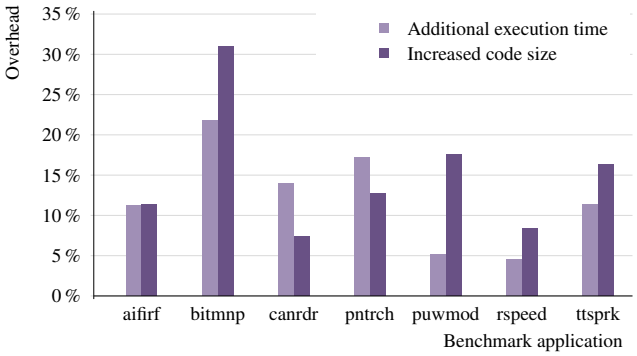


Figure 6. Overhead of EEMBC benchmarks

E. Overhead

The software instrumentation of our technique provides a higher level of reliability but causes overhead. We aim to find an optimal trade-off between execution time and memory overhead on the one hand and good results concerning error detection on the other.

As described in Section IV-B, our instrumentation technique needs three processor instructions for each checkpoint. Since the CarCore is able to execute two of these instructions in parallel, the execution of a checkpoint usually requires two processor cycles. Fig. 6 shows the results measured on the selected EEMBC benchmarks; as can be seen, the additional execution time is low, only 12.2 % in the average case. To determine the memory overhead we compare the number of instructions with the original benchmark program without instrumentation. Here, we can see an increased code size of 15.0 % in the average case. Regarding the benchmark *bitmnp*, we observe slightly higher results, since this application contains lots of very small basic blocks.

VI. Summary and Future Work

In this paper, we have presented the fault detection capabilities of our hybrid hardware-software technique for the on-line detection of control flow and timing errors. Our approach goes one step beyond related checking mechanisms:

 ISBN: 978-1-61208-233-2

Besides monitoring only logical correctness of the control flow, we additionally introduce a technique to guarantee temporal correctness, especially focusing on hard real-time systems.

We have implemented our error detection technique for the hard real-time capable CarCore processor. The hardware overhead of the check unit is very low, it requires only 0.5 % of ALUTs compared to the processor core. Fault injection experiments on automotive benchmarks prove the effectivity of our approach: More than 65 % of injected SEUs uncaught by processor exceptions can be detected. The number of simulation runs terminating with wrong results can be reduced by more than 60 %, the rate of endless loops even by 90 % using the proposed mechanism. Furthermore, the detection latency of our technique is very low: An error is detected after only 22.1 cycles in the average case. Moreover, we measured the instrumentation overhead for several benchmarks. In our evaluations, the mean additional execution time is only 12.2 %, while the increased code size is around 15.0 %.

In our future work, we will further optimize the ratio between coverage, latency and the occurring overhead: In case of a long basic block with a high WCET, a potential fault is currently detected with high latency. This problem can be avoided by splitting blocks and adding extra checkpoints in the middle. On the other side, very small basic block causing much overhead could be combined with neighboring blocks without suffering from detection quality.

REFERENCES

[1] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham, "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection," *IEEE Trans. Par. and Dist. Systems*, vol. 10, no. 6, pp. 627–641, 1999.

[2] "ALTERA Stratix II Device Handbook, Volume 1 (ver 4.5)," http://www.altera.com/literature/lit-stx2.jsp, 2011.

[3] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An Open Toolbox for Adaptive WCET Analysis," in *Proc.* 8^{th} *SEUS Workshop*, 2010, pp. 35–46.

[4] P. Bernardi, L. Bolzani, M. Rebaudengo, M. S. Reorda, F.L.Vargas, and M. Violante, "A New Hybrid Fault Detection Technique for Systems-on-a-Chip," *IEEE Trans. Comp.*, vol. 55, no. 2, pp. 185–198, 2006.

[5] A. Bonenfant, I. Broster, C. Ballabriga, G. Bernat, H. Cassé, M. Houston, N. Merriam, M. de Michiel, C. Rochange, and P. Sainrat, "Coding Guidelines for WCET Analysis Using Measurement-Based and Static Analysis Techniques," IRIT Toulouse, Tech. Rep., 2010.

[6] E. W. Czeck and D. P. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," in *Proc.* 20^{th} *Int'l Symp. Fault-Tolerant Computing (FTCS)*, 1990, pp. 236–243.

[7] "EEMBC AutoBench 1.1," http://www.eembc.org/, 2011.

[8] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-Error Detection Using Control Flow Assertions," in *Proc.* 18^{th} *IEEE Int'l Symp. Defect and Fault-Tolerance in VLSI Systems (DFT)*, 2003, pp. 581–588.

[9] HighTec EDV-Systeme GmbH, http://www.hightec-rt.com/.

[10] Infineon Technologies AG, *TriCore 1 User's Manual*, January 2008, v1.3.8.

[11] D. Lu, "Watchdog Processors and Structural Integrity Checking," *IEEE Trans. Comp.*, vol. 31, no. 7, pp. 681–685, 1982.

[12] A. Mahmood and E. McCluskey, "Concurrent Error Detection Using Watchdog Processors–A Survey," *IEEE Trans. Comp.*, vol. 37, no. 2, pp. 160–174, 1988.

[13] J. Mische, I. Guliashvili, S. Uhrig, and T. Ungerer, "How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT," in *Proc.* 23^{rd} *Int'l Conf. Architecture of Computing Systems (ARCS)*, 2010, pp. 2–14.

[14] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow Checking by Software Signatures," *IEEE Trans. Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[15] J. Ohlsson and M. Rimen, "Implicit Signature Checking," in *Proc.* 25^{th} *Int'l Symp. Fault-Tolerant Computing (FTCS)*, 1995, pp. 218–227.

[16] M. Paolieri and R. Mariani, "Towards Functional-Safe Timing-Dependable Real-Time Architectures," in *Proc.* 17^{th} *Int'l On-Line Testing Symp. (IOLTS)*, 2011, pp. 31–36.

[17] R. Ragel and S. Parameswaran, "A Hybrid Hardware–Software Technique to Improve Reliability in Embedded Processors," *ACM Trans. Embedded Comp. Systems*, vol. 10, no. 3, pp. 36:1–36:16, 2011.

[18] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance," in *Proc. Int'l Symp. Code Generation and Optimization (CGO)*, 2005, pp. 243–254.

[19] M. Schuette and J. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Trans. Comp.*, vol. 36, no. 3, pp. 264–276, 1987.

[20] L. Thiele and R. Wilhelm, "Design for Timing Predictability," *Real-Time Systems*, vol. 28, no. 2, pp. 157–177, 2004.

[21] "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[22] K. Wilken and J. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors," *IEEE Trans. Comp.-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 6, pp. 629–641, 1990.

[23] J. Wolf, B. Fechner, S. Uhrig, and T. Ungerer, "Fine-Grained Timing and Control Flow Error Checking for Hard Real-Time Task Execution," in *Proc.* 7^{th} *Int'l Symp. Industrial Embedded Systems (SIES)*, 2012, pp. 257–266.

[24] J. Wolf, B. Fechner, and T. Ungerer, "Fault Coverage of a Timing and Control Flow Checker for Hard Real-Time Systems," in *Proc.* 18^{th} *Int'l On-Line Testing Symp. (IOLTS)*, 2012, pp. 127–129.

 ISBN: 978-1-61208-233-2

Software Validation

When 'Pure Mathematical Objectivity' is no Longer Enough

Isabel Cafezeiro
Instituto de Computação
Universidade Federal Fluminense
Niterói/
Programa de História das Ciências e das Técnicas e
Epistemologia (HCTE-UFRJ)
Brasil
e-mail: isabel@dcc.ic.uff.br

Ivan da Costa Marques
Programa de História das Ciências e das Técnicas e
Epistemologia (HCTE-UFRJ)
Universidade Federal do Rio de Janeiro
Rio de Janeiro,
Brasil
e-mail: imarques@ufrj.br

Abstract— **By focusing on systems that can be trusted to operate as required, software validation offers a rich field to study how far one can go with the support of mathematical certainty, that is, to identify when evidence (a non formal entity) must come into play to dismiss the possibility of critical errors. First, this article highlights that the view of mathematics as a source of accuracy supported by a purified and rational chaining of reasoning persists until the present days. Resorting to historical controversies of the 1970's regarding software validation, it is possible to indicate local (social) elements that necessarily participate in what is usually considered 'technical' or 'objective', showing therefore that there is no way to establish rigid or fixed boundaries delimiting what is considered 'exact'. Regarding software correctness, the sociotechnical approach adopted in this paper leads to a intertwined frame where social (collaborative) mechanisms act in ways that are inseparable from those mechanisms that are considered 'technical' or 'objective', which are, in this case, formal methods. This paper discusses software validation in the light of Sociology of Mathematics and Social Studies of Science and Technology.**

Keywords- formal specification; collaborative development; objectivity; sociology of mathematic.

I. INTRODUCTION

'The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness' [1]. Edsger Dijkstra, a spokesman of formal methods for software reliability in the seventies, argued in favor of a more rigid way to develop software, as a reaction to the just before denounced *software crisis* in the 1968 Conference on Software Engineering, Garmisch, reported in [2]. He defended that programs should be constructed 'hand-in-hand', module by module, with their formal proof. Dijkstra opposed his proposition to the traditional technique, 'to make a program and then to test it', which, in his view, was an effective way to detect the presence of errors but did not guarantee their absence. [1]

Dijkstra allied himself with a powerful partner: mathematics. Among mathematicians, and also in the common sense, there is a widespread culture of objectivity and accuracy of mathematics [3], which is strong enough to stifle dissenting voices, those sympathetic to non-formal mechanisms [4].

A. *The strengthening of mathematics to support the trust*

'Why are mathematical certainty and the evidence of demonstration common phrases to express the very highest degree of assurance attainable by reason?' [5]

It is not the purpose of this section to present an exhaustive historical account of the extensively expanded relations between mathematics and trust and certainty. Much more modestly, it brings in for discussion some historical moments over the last few centuries where those relations were under debate. The above quotation is a milestone: John Stwart Mill, in 'A System of Logic' (mid-nineteenth century) took a stand against the association of the highest degree of safety reachable by reason 'to mathematical certainty' and 'the evidence of demonstration'. This triggered intense objection by Gottlob Frege (Die Grundlagen der Arithmetik – 1884) [6], today considered one of the founders of modern logic, a spokesman for the strengthening of rationalist trend. The 'peculiar certainty' attributed to so-called 'deductive science' gained momentum in early twentieth century, through the Vienna Circle, where the logical positivists, declared their rejection to what they called theological and metaphysical speculation [7]. At the same time, amid the movements of mathematical foundations, in particular, David Hilbert's formalist program came to the fore. It conceived mathematics as a purely formal system, consisting of symbols devoid of meaning or interpretation: 'In a sense, mathematics has become a court of arbitration, a supreme tribunal to decide fundamental questions — on a concrete basis on which everyone can agree and where every statement can be controlled.' [8] The 1930s revealed surprises to these approaches, especially to the formalist program, with the publication of Gödel Theorems [9], which demonstrated the existence of statements that, although they could be written in a formal system of a certain kind, could not be proved in it. This exposed the inability of mathematics to decide any mathematically expressed matter maintaining its consistency. The publication of Gödel's theorems put into question the role of mathematics as a 'court of arbitration' as envisioned in the Hilbert's formalist program. Moreover, in

 ISBN: 978-1-61208-233-2

the 1940s some mathematicians have realized the need to consider factors then considered 'extra-mathematical' for understanding the configuration of mathematics itself. For example, the Dutch mathematician Struik proposed a 'Sociology of Mathematics' to be concerned 'with the influences of forms of social organization on the origin and growth of mathematical conceptions and methods' [10].

However, it is noteworthy that in the 1970s, Dijkstra proposed a program of mathematization of software by mobilizing arguments in bases that were very similar to those that David Hilbert had proposed in the formalist approach: the effort in the pursuit of mathematical truth and accuracy and consistency of mathematical methods. Even today, it is to be highlighted, the discourse of the search for security and reliability is widely supported by confidence in mathematics and formal systems. An illustrative example can be found in the general terms that conducted the formulation of 'The Grand Challenge Project' in 2005 attesting an enthusiastic view of formal methods: 'Programmers of the future will make no more mistakes than professionals in other disciplines. Most of their remaining mistakes will be detected immediately and automatically, just as type violations are detected today, even before the program is tested. An application program will typically be developed from an accurate specification of customer requirement; and the process of rational design and implementation of the code will be assisted by a range of appropriate formally based programming tools, starting with more capable compilers for procedural and functional programming languages.' [11]

The Sociology of Mathematics [12][10] allows us to question the 'objectivity' that is sought in mathematics and formal methods, highlighting that in its own conformation, mathematical entities are inseparably mixed with local or temporal elements, and are therefore historically situated. Such viewpoint takes social mechanisms for collaboration on a par with formal methods in the validation of software systems and reinforces the role of inductive reasoning, tests, empirical approaches as allies in the process of software validation.

B. Organization of the next sections

In Section II, we analyze the spreading of formal methods as a guarantee of software correctness since the 60's until today. In this process, we point out the inevitable presence of personal choices and subjectivities that the formal methods are unable to eliminate and the power of speech that relies on a so-called 'objectivity' of mathematics to enhance the trust in formal methods and the promise of a software free of errors.

In Section III, we present a case study relating mathematics and computers that reinforces the view that, when mathematics is requested to be applied in real-world situations, not only local issues are modified as a result of interaction with mathematics, but also mathematics changes as result of interactions with local issues. This view collapses with the general conception that math is unique and immutable as a 'language of Gods', a conception that persists not only in common sense, but also among mathematicians by cultural reasons. As expressed in David Hilbert's Radio Broadcast, in 1930: 'Already Galileo declared: "To understand nature, we must learn the language and the signs through which nature speaks to us." But this language is mathematics, and these signs are mathematical figures!'

In the case analyzed here, we report a new arithmetic - one that is remodeled by requirements of a computer hardware, what shows that in mathematics there is room for subjectivity. This example is a contribution to the understanding of how social elements come to be inseparable from the setting of mathematics, becoming part of it.

Then, in Section IV we return to the subject of software correctness. We consider arguments that emerged in the 1970s, in response to the mathematization of software. These reactions emphasized the importance of considering social mechanisms in the development of secure software and software verification. These social mechanisms gain a new dimension when we consider the new capabilities of interaction provided by the Internet, new techniques of software development considering collaboration and reuse and the speed of technology nowadays. Furthermore, we also consider a recent testimony in favor of the association between formal methods and empirical mechanisms. These are allies with whom formal verification can go far beyond.

We conclude this article indicating the Sociology of Mathematics and Social Studies of Science and Technology as emerging areas that consider the interweaving of mathematical knowledge and social and subjective issues. This kind of research supports mixed approaches in which recent mechanisms of collaboration can be taken in to build solutions to problems that were previously treated as purely technical.

II. Formal Methods for Software Correctness (from the seventies to today)

It was in terms of trust in mathematics in the late 1960s and early 1970s, when computer programs had become too long and were used in applications involving safety-critical situations, that the U.S. Department of Defense (DoD) initiated a series of debates that pointed to the mathematization of systems as a guarantee of correctness. At that time, the aim was to create a systematic methodology for building systems, to question the effectiveness of empirical tests and to bet on formal specifications as a means to enable secure programming in two ways: first, since the specification languages are more ‘abstract’ (more distant from the code that activates the hardware) than the programming languages, they can be closer to the problem domain, thus facilitating the correct understanding and ease of expression of the solution; and, second, since the specification languages are formal, they would be suitable to prove properties of programs, ensuring correctness. In 1985, the U.S. DoD published the Orange Book whose 'purpose [was] to provide technical hardware/firm-ware/software security criteria and associated technical evaluation methodologies', mandatory for use by all DoD Components in carrying out ADP (Automatic Data Processing) system. [13]

The establishment of these kinds of standards continued. In 1999 an arrangement of international organizations called Common Criteria (CC) created a basis for evaluating the security of information technology products, which then replaced the Orange Book. The CC defined seven levels of

 ISBN: 978-1-61208-233-2

assurance, (EAL), establishing a degree of trust directly proportional to an adherence to formal methods:

TABLE I. The Common Criteria Evaluation Assurance Levels: the more formal, the more reliable.

EAL1: Functionally Tested,
EAL2: Structurally Tested,
EAL3: Methodically Tested and Checked,
EAL4: Methodically Designed, Tested, and Reviewed,
EAL5: Semiformally Designed and Tested,
EAL6: Semiformally Verified Design and Tested,
EAL7: Formally Verified Design and Tested.

The role of formal methods in the view of Common Criteria can be understood from the report [14]: to earn certification the developer chooses and formalizes the properties he considers indispensable for safety, provides a formal specification of the parts of the software he considers critical and a proof that the chosen properties meet the specification. The last step is then to prove that the program is indeed a refinement (an implementation) of the given specification, and thus meets the properties proved at the formal level. These documents are then analyzed by the 'evaluation authority' – a team of specialists whose name reveals the sense of authority provided by mathematics.

As an example, we refer to [15], which describes 'how formal methods were used to produce evidence in a certification, based on the Common Criteria, of a security-critical software system'. This experience report makes clear that even being extremely formal the process always starts from choices, and these are inevitably subjective. As this report shows, the software developer chooses the pieces of code that are 'security-relevant software behavior'. He also decides which are the properties to be proved and where to locate the preconditions and postconditions in the code. However, what is considered difficult in the certification process are the formal steps, while the developer's choices are only briefly mentioned: 'Given 1) source code annotated with preconditions and postconditions and 2) a security property of interest, the overall problem is how to establish that the code satisfies the property. We developed a five-step process for establishing the property. These five steps are described (...)'.

As one might expect, arbitrariness, convention, and hence 'subjectivity' are inevitably present in the initial stages, when several choices are made by the developer. The formal method is unable to eliminate subjectivity, but propagates it stealthily throughout the entire process.

Ignoring the subjective character of choices such as those pointed out above, and still evoking the absolute certainty in formal methods, we can see nowadays in the CC web site statements such as: 'IT products and protection profiles which earn a Common Criteria certificate can be procured or used without the need for further evaluation' suggesting that formal methods are reliable enough to bypass the need of any additional testing, and overshadowing that its role is to provide strong evidence that the system does not contain critical errors.

III. When objectivity is no longer enough

In software verification and validation, the criterion of truthfulness, reliability and applicability is many times dependent upon confidence in proofs, which, in turn, has been historically linked to the purely deductive reasoning (or 'the ideal of certainty achieved by mathematical proof', in words of Hoare [16]). This does sound a bit contradictory since, in computer science, the abstract (formal) knowledge becomes directly embodied in computer programs, and so, apparently makes direct contact (without intermediation) with the 'life-world', that is, borrowing the term from Edmund Husserl, the 'only real world, the one that is actually given through perception, that is ever experienced and experienceable - our every-day life-world' [17]. Thus, inductive reasoning, tests, empirical approaches as well as methodologies based on social collaboration appear in programming activities side by side with formal methods, as a way of approaching 'our every-day life-world'. Computer programming evinces that knowledge is a situated construction, that is, a construction strongly connected to materialities and local issues, even when the subject is mathematics or other technical or abstract subjects.

Let us consider the controversies around the establishment of the IEEE Standard for Floating-Point Arithmetic [18]. It shows the conflict between the 'objective' arithmetic and the requirements of 'every-day life-world', here embodied in a hardware architecture. In this struggle, both sides change, giving rise not only to a modified computer but, as a counterpoint of Frege's claim ('there is nothing more objective than the laws of arithmetic' [6]) giving rise also to a modified arithmetic.

The core issue was the confrontation of the infinite expansion of certain real numbers and the finite size of computational representation, which certainly requires some form of truncation. Different algorithms were used by different companies (IBM, Digital, HP, Intel, Texas) which generated different results for the same purpose. A comparison between them showed that there were many decisions to be taken: 'One specialist cite[d] a compound-interest problem producing four different answers when done on calculators of four different types: $331,667.00, $293,539.16, $334,858.18 and $331,559.38. He identifie[d] machines on which a/1 is not equal to a (as, in human arithmetic, it always should be) and $e\pi - \pi e$ is not zero.' [18]

The total number of digits to be adopted in the computational representation of real numbers was a decision that involved the complexity of the hardware being used. In addition, other decisions would also influence hardware design, such as the size of the sequence of digits for representing the mantissa and exponent in the floating point representation. Mackenzie [18] also pointed out the mathematical arbitrariness embedded in several choices: what should be done if the result of a calculation exceeds the largest absolute value expressible in the chosen system, or if it falls below the lowest? What should be done if a user attempts a meaningless arithmetic operation such as dividing zero by zero? In addition to producing different results in some calculations, the lack of standardization hindered compatibility among different computers. Thus, it was necessary to define a standard computational arithmetic.

 ISBN: 978-1-61208-233-2

However, changes in the numerical representation implied costly hardware changes and other nuisances such as lack of compatibility with preexisting programs. Fundamental questions persisted: these different forms of representation configured a new arithmetic or were they just different ways of representing the sole real arithmetic?

'Negotiating arithmetic', as Mackenzie aptly termed it, proved to be a long process. A committee began to work in 1977 but the norm IEEE 754, Numbers Fractional Binary Arithmetic, was not adopted until 1985. The crucial point, highlighted by [18], is that 'there was a stable, consensual human arithmetic against which computer arithmetic could be judged. Human arithmetic was, however, insufficient to *determine* the best form of computer arithmetic. (…) Human arithmetic provided a resource, drawn on differently by different participants, rather than a set of rules that could simply be applied in computer arithmetic.'

IV. Social Processes for Software Correctness

Despite the strength of mathematization of software, even in the seventies the confidence in formal methods was not a consensus: '[L]et us suppose that the programmer gets the message 'VERIFIED.' (. . .) What does the programmer know? He knows that his program is formally, logically, provably, certifiably correct. He does not know, however, to what extent it is reliable, dependable, trustworthy, safe; he does not know within what limits it will work; he does not know what happens when it exceeds those limits. And yet he has that mystical stamp of approval: "VERIFIED." ' [19]. Hence, subjectivity was clearly pointed out, but was insufficient to shake the confidence that rested solely on formal methods, and even applies today.

The dissenting voices did more than point out the existence of a social component in the acceptance of theorems and proofs. They argued that it is precisely the social component that acts as a decisive factor of trust and may lead to minimize the error conditions: 'What elements could contribute to making programming more like engineering and mathematics? One mechanism that can be exploited is the creation of general structures whose specific instances become more reliable as the reliability of the general structure increases. This notion has appeared in several incarnations, of which Knuth's insistence on creating and understanding generally useful algorithms is one of the most important and encouraging. Baker's team-programming methodology is an explicit attempt to expose software to social processes. If reusability becomes a criterion for effective design, a wider and wider community will examine the most common programming tools.'[19].

Although these ideas have not strongly echoed then, we now see that 'expose software to social processes' seems to be the trend in the development of secure software. Researchers cited generally useful algorithms that took the form of the present design patterns which are general descriptions of how to solve a commonly occurring problem in software design. A pattern is an unfinished algorithm that must to be adapted to many real situations. They also cited team programming methodologies that nowadays have been improved by the collaborative capabilities introduced through the Internet, in a way that a piece of code can be constructed or examined by several hands, tending to stability. Reusability is a key issue in the conception of modern program environments, enabling stable codes to be used as components in the construction of modules. Finally, there is currently a proliferation of software development methodologies which rely on social collaboration for secure software development. Some examples are TDD, PBL, social coding, pair programming. Test Driven Development (TDD) is a programming methodology where any functionality of a program starts from a failing test case. Each piece of code is written to solve the test case and overpass its failure. Problem Based Learning (PBL) is a learning methodology that considers a realistic problem, with all its complexity, as a way of invoking interdisciplinarity and autonomy aiming at knowledge construction through the design and implementation of a solution for the proposed problem. Both TDD and PBL takes place in teams. Social Coding are face events aiming the development or enhancement of code in groups. Pair Programming is a programming mechanism guided by a "pilot" trading ideas with a "co-pilot" attended by an audience. Each of these methodologies, among others, start from the assumption that the collective creation, negotiation, discussion and review by multiple agents, among other mechanisms of participation, tend to maximize the chances of success in building a product, especially software.

V. Conclusion

In the Strong Program in the Sociology of Knowledge of the University of Edimburgh, case studies play an important role as they bring in the complexity of the 'life-world' situations. The Sociology of Mathematics, a sub-area of the Strong Program, is a field were the resistance against a intertwined approach to mathematics is a key issue of study, and case studies make visible this resistance. David Bloor, one of the proponents of the Strong Program, referring to questions involving the myth of a purified mathematics, claimed that '[t]he best answer to these questions is to provide examples of such sociological analyses' [12].

In the line of the Sociology of Mathematics, this article pointed out that strategies of software certification currently in place and the tone of recent initiatives such as 'The Great Challenge' indicate that confidence in purified mathematics still underlies the thinking and the doing of mathematicians and computer scientists. In the sequence, this article brings in a case study concerning the definition of an arithmetic for computers that makes a compelling argument about the impossibility of achieving a purified arithmetic, that is, an arithmetic that is not influenced by what is considered 'extra-mathematical' factors. In a attempt to have purified mathematics as an arbiter, new elements and testimonies have slowly emerged, destabilizing the bases of the search for objectivity and making room for mixed heterogeneous (extra mathematical) elements that were decisive in establishing consensus. We thus conclude this paper by citing two recent cases that argue in favor of hybrid approaches, rejecting the possibility of a mathematics that, being supposedly free of subjectivities, would provide absolute certainty.

 ISBN: 978-1-61208-233-2

The first case is a recent discussion referring to a famous phrase of the mathematician Georg Cantor: '*Je vois mais je ne le crois pas*', by which he would have expressed his astonishment at the amazing results that he had just discovered. According to the analyses of the mathematician Gouvêa [20], however, this phrase was actually a response to Dedekind who argued contrary to Cantor's proposals. It was an emphatic trope against his opponent's arguments about something that was, for Cantor, completely clear. Gouvêa's conclusions about this case have much to do with the discussion of objectivity and the influences of non-mathematical factors in the configuration of what is said to be objective. According to Gouvêa (in page 198) '[t]he story was then co-opted to demonstrate that mathematicians often discover things that they did not expect or prove things that they did not actually want to prove.'

Sociology of Mathematics argues that mathematical knowledge is a result of several steps of agreement within a collective thought, in strong alignment with Gouvêa's assertion about subjectivity in mathematical proofs: 'A proof is not a proof until some reader, preferably a competent one, says it is. Until then we may see, but we should not believe.'

The second case is about a recent statement of Tony Hoare, a well known knowledgeable spokesman, for the use of formal methods to ensure program correctness. As late as 2010 Hoare felt adequate to announce a reconsideration of his own previous words. In page 5 of [21]: 'I regarded program testing as the main rival technology'. He reported a join work where he could see senior researches using formal methods not for proof but to detect program errors as much close as possible to their occurrence in code. He then concludes: 'Testing and proving are not rivals: they are just two ends of a scale of techniques available to the software engineer to collect evidence for the validity and serviceability of delivered code.' Hoare's testimony in a year as recent as 2010 shows for how long inductive reasoning, tests, empirical approaches have been (and very likely still are) rejected as legitimate mechanisms for software verification.

The Sociology of Mathematics bring legitimacy to explanations of mathematical facts (such as proved theorems) which distance themselves from explanations of a more absolutist flavor prevailing among the majority of mathematicians. For the Social Studies of Science and Technology, where the universality of knowledge is understood as a mechanism to ensure authority and science is viewed as a local phenomenon, objectivity is addressed in its interweaving with the social; this makes it possible that other elements besides those considered as 'technical' come into play in the composition of the facts regarded as 'mathematical'. A closer examination of mathematical practice shows this inevitable interweaving of knowledge, which, however, remained invisible to the great majority possibly because of the lack of interest in shaking the stability of a purified body of knowledge which places mathematics in a level of unquestionable, neutral and universal truth.

References

[1] E. W. Dijkstra, "The humble programmer." Commun. ACM, vol 15, issue 10, pp. 859-866, October 1972.

[2] P. Naur and B. Randell, Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, pp. 7-11 October 1968.

[3] T. M. Porter, Trust in numbers: the pursuit of objectivity in science and public life, Princeton University Press, 1995.

[4] D. A. Mackenzie, Mechanizing proof: computing, risk, and trust. Cambridge, Mass. MIT Press. 2001.

[5] J. S. Mill, A System of Logic, Raciocinative and Inductive, Londres, H& B Pub, 1848.

[6] G. Frege, The Foundations of Arithmetic. A logico-mathematical enquiry into the concept of number, Harper & Brothers. New York. 2Ed, 1953.

[7] R. Carnap, O. Neurath, and H. Hahn, "La concepción científica del mundo: el Círculo de Viena. 1929" Trad. Lorenzano, P. Presentación de La concepción científica del mundo: el Círculo de Viena. Revista Redes, vol.9, n. 18, pp. 103-149,2002..

[8] D. Hilbert, "On the infinite", In: Bencerraf, P., Putnam, H. Eds. Philosophy of Mathematics Cambridge University Press, 1984.

[9] M. Davis, The undecidable; basic papers on undecidable propositions, unsolvable problems and computable functions, Raven Press Hewlett, N.Y., 1965.

[10] D. J. Struik, "On the Sociology of Mathematics", Science and Society, New York, vol. VI, no. 1, Winter, 1942.

[11] C.A.R. Hoare and J. Misra, "Verified Software: Theories, Tools, Experiments Vision of a Grand Challenge Project". In B. Meyer, J. Woodcook, eds. First IFIP TC 2/wg 2.3, VSTTE 2005, Zurich, vol 4171 of LNCS pp. 1-18, Springer, 2005.

[12] D. Bloor, Knowledge and social imagery. Chicago: University of Chicago Press, 1976/1991.

[13] Department of Defense Standard. Department of Defense Trusted Computer System Evaluation Criteria. DoD 5200.28-STD, Dec, 1985 av. at http://csrc.nist.gov/publications/secpubs/rainbow/std001.txt [retrieved:September,2012]

[14] Common Criteria for Information Technology Security Evaluation Part 3: Security assurance components July 2009 Version 3.1 Revision 3. CCMB- 2009-07-003:229.

[15] C. L. Heitmeyer, "On the role of Formal Methods", Electronic Notes in Theoretical Computer Science, vol 238, pp. 3–9, 2009.

[16] C.A.R. Hoare, "How did software get so reliable without proof?" FME'96: Industrial Benefit and Adv. in Formal Methods, vol. 1051 , pp. 1-17, 1996.

[17] E. Husserl, The crisis of European sciences and transcendental phenomenology; an introduction to phenomenological philosophy. Evanston: Northwestern University Press, 1954/1970.

[18] D. A. Mackenzie, Negotiating Arithmetic, Constructing Proof. In: D. Mackenzie Ed. Knowing Machines - Essays on Technical Change. Cambridge, MA: The MIT Press, 1996.

[19] R. A. De Millo, R. J. Lipton, and A. J. Perlis, Social Processes and Proofs of Theorems and Programs, Commun. ACM vol 22, pp. 271-280, 1979.

[20] F. Q. Gouvêa, "Was Cantor surprised?", The Mathematical Association of America, Monthly vol 118, pp. 198-209, 2011.

[21] C. A. R. Hoare. "Testing and proving, hand-in-hand" TAIC PART'10, L. Bottaci and G. Fraser Eds. Springer-Verlag, Berlin, Heidelberg, pp. 5-6, 2010.

 ISBN: 978-1-61208-233-2

A Software Quality Framework for Mobile Application Testing

Yajie Wang, Ming Jiang, Yueming Wei
China Telecom Corporation Limited Beijing Research Institute
Beijing, China
Email: {wangyj, jiangming, weiym}@ctbri.com.cn

***Abstract*-With the explosion of mobile applications, all application providers expect to work out a popular mobile service. There are two features of a popular mobile service: adapting for mobile device's diversity and achieving high user satisfaction. For Quality Assurance (QA) testers, the former feature brings heavy testing workload and the latter claims testers try their best for good quality and good usability of the service. Therefore, improving work efficiency and test completeness are critical for mobile application QA testers. In this paper, a test framework for mobile applications is proposed, which aims to help QA testers work with high efficiency and contribute to good products with nice user experience. Moreover, a case applying this framework is presented for validating it.**

Keywords-Quality assurance; QA Tester; Mobile application; Usability

I. INTRODUCTION

At present, with the development of wireless networks and the popularization of mobile devices, mobile applications become more and more popular [1][2]. The traditional desktop software developers are putting considerable effort into the development of the mobile applications gradually. Also, telecom operators are caring more about the increase of business profits obtained from mobile applications and try their best to seek some "killer" mobile applications. With the rapid growth of mobile applications market, demands on software quality rises rapidly. The applications are expected to be stable, be quick response and have good UI experiences [3][4]. To satisfy these requirements, project team members, including software designer, developer, tester, project leader and QA member [5] should work together. Everyone should take special care of the characteristics of mobile applications and assure the typical quality of them in their working phase. In this paper, we focus on mobile software quality [5][6] only from the view of test and validation.

Most test concepts and principles of desktop software can be adopted in mobile application testing [2]. However, there are some obvious differences between software for mobile devices and desktop software [3]. The characteristics of mobile device and the complex application scenario of using applications cause the difference. As another point of view, adapting for mobile device's diversity and achieving high user satisfaction [4] are critical factors of a successful application. In desktop software, the PC, browser, connection, and context of use are so standard that even researchers do not realize or remember to mention them affecting software quality and user experience [7]. The traditional test schema is not appropriate. There should be new approaches and concerns fitting to these differences. Therefore, as QA testers, we make extra emphasis on these aspects: the mobile devices features and diversity, usage scenario in real life and user experience.

This paper proposes a systematic framework for improving software quality of mobile applications by analyzing the characteristics from all its aspects and from multiple perspectives. This paper is comprised of five sections. In Section II, we give an overview of the framework and make a brief description of its components. Section III is dedicated to describing the implementation of the components. In Section IV, a case study is presented to valid the framework. Section V concludes this paper with a summary and outlines the field of research for future work.

II. TEST FRAMEWORK ARCHITECTURE

An optimal quality assurance system for mobile software means to work in an accurate and efficient way, and to submit products with high user satisfaction. The framework proposed in the paper is composed of four components: a mobile devices information system, a defect system for mobile applications, an aggregation of key test scenarios and a mechanism for usability test.

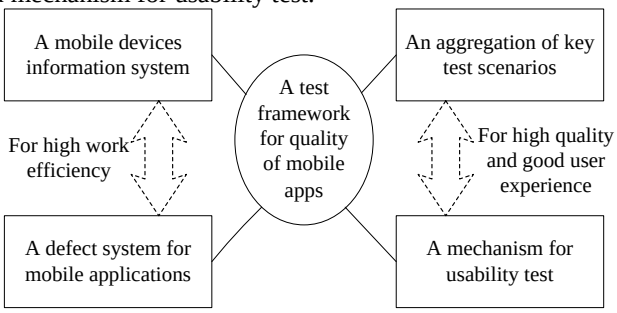


Figure 1 Architecture of the framework

The first two parts contribute to providing an accurate and efficient working condition. The mobile devices information system includes a device database and a real device management system. The database and the management system can be accessed by the whole team members. To QA testers, the database is an effective support to choose test objects and to make contrasting test plan for different devices; at the same time, the real device management system assists testers to find available devices

 ISBN: 978-1-61208-233-2

as early as possible. The defect system for mobile applications is distinguished from an ordinary defect system. It defines some special types of defects and some special attributes of a defect, which are peculiar to mobile applications. QA testers apply the defect system to accurately describe defects they found and then make it easy to be understood and dealt with among every team of the project.

The last two parts are dedicated to giving users high satisfaction. According to the characteristics of mobile device, the complex usage scenario of mobile applications and the problems easy to be neglected in mobile software testing, an aggregation of key test scenarios is defined. It collects five parts: test scenarios related to resource limitation [3], test scenarios related to imitating real usage activities, test of the server portion of the application, test of those related to charge, privacy and legacy [3], and test for good user experience. Only a mobile application verified from these five aspects can be called a valid application, not just being a software meeting service logic. A mechanism for usability test [4] describes an effective way to have a usability test. It is used by QA tests to gain usability challenge and advice from outside the application's working team, most of which are greatly valuable contributions.

III. IMPLEMENTATION OF THE FRAMEWORK

In this section, the implementation of four components comprising this framework is described in detail.

A. Structuring a Mobile Devices Information System

Diversity of mobile devices makes great difference to mobile application development and test. Unlike traditional desktop software, a good mobile application should be adapted for various devices. Large amount of work was spent on the adaption. Creating a device database to keep track of device information is a great way to improve work efficiency.

Here "database" generally refers to anything from a Microsoft Excel spreadsheet to a little SQL database [3], or any other software or system, if practicable. Scale of the database can vary according to the company's scale and cost. Devices information can be stored in the database during the requirements phase of a project or later as a change in project scope [3]. A record for a mobile device should at least include the following items:

- Important device technical specification details (screen resolution, OS version, hardware details, supported media formats, input methods, localization, any optional features, etc.)
- Any firmware upgrade or modification information, especially those related to hardware modifications.
- Any known bugs and important limitations with the device.

In addition, the information of how to get actual testing device (such as available from real device library, purchased or loaned through manufacturer or carrier loaner programs) is suggested to be recorded in the database.

For real device management, two aspects are highlighted. One is implementing a library check-in and check-out system. Team members can reserve devices for testing and development purposes. It facilitates sharing devices across teams, and then improves work efficiency greatly. The other is defining what device is a "clean" device [3] and how to return to the same starting state. At present, there is no good way to "image" a device; however, it is a basic testing policy for QA testers. There are some common ways, such as a specific uninstall process, some manual clean-up, or sometimes a factory reset. If the detailed operation steps are recorded, testers will save much time for learn and trial.

So, how do QA testers use the device information system? First, they analyze the similarity of devices and divide them into different groups, for example, grouping by the platform OS. Next, they choose the target devices according to the actual project and make test plan. Besides primary function, test priority and special test scenarios for every device should be included in the test plan. Then, QA testers use the real device management system to get the real devices rapidly. Finally, execute the test.

B. Building a Defect System for Mobile Applications

Almost all defects for desktop software may occur on mobile applications. Some typical defaults are program crashing and unexpected terminations, inadequate input validation, features not functioning expected, responsiveness problems and poor usability issues. We redefine the term defect for mobile applications from a larger range which not only includes these typical defects. Some types of defects typical on mobile applications are highlighted:

- Using too much disk space/memory on the device, not releasing memory or resources appropriately.
- Usability issues related to input methods, font sizes, and cluttered screen real estate. Cosmetic problems that cause the screen to display incorrectly [3].
- Application "not playing nicely" on the device [3], such as not compatible with other applications, overusing network resources, incurring extensive user charges, etc.
- Not handling private data securely. This includes not ensuring data safety of mobile device and server, or not guaranteeing safety of data transmission on network.
- Application not conforming to the third-party agreements, such as Android SDK license agreement (if involved), Google Maps API term (if involved), or any other terms if applied to the application.

Most defect tracking systems can be customized to work for the test of mobile applications. Whatever a system is adapted; there are some important defect attributes to be encompassed in the system, for the purpose of clarifying a mobile software defect. These attributes are:

- The application version information, language, and so on.
- Device configuration and state information including device type, platform version, network state, and carrier information.
- Steps to reproduce the problem. The steps should be described exactly using predefined standardized

 ISBN: 978-1-61208-233-2

terms, such as clear versus back, click versus tap, and so on.

- Device screenshots, which can be taken through screenshots software developed for mobile devices.

C. Defining an Aggregation of Key Test Scenarios

1) Resource limitations of mobile applications

Although mobile devices experience a massive gain in performance in recent years [2], resource limitation is still a topic we have to talk about. These limitations include devices limitations and network limitations. Devices limitations vary in terms of memory, processing power, screen type, battery level, storage capacity, platform version, input method, etc., network limitations vary in terms of accessibility and bandwidth.

To stand-alone applications, most core functions run in local memory, so testing of these applications often focuses on the limitations of the device itself. To network-driven applications, it provides a lightweight client on the device but relies on the network to provide a good portion of its content and functionality, so besides devices limitation, we also should focus network limitations when testing of these applications.

2) Imitating real usage activities

QA testers should try doing anything impossible on the mobile device, or imitating some "strange" activities when testing the application. We divide these activities into two aspects: whether compatible with other programs and how to deal with accidents.

In the real world, your application is only one of many installed on the device. You should check whether the software works well together with other device functions or applications. You should consider many things. Will your application rely on other service or content provider? Will your application act as a service or content provider? After all, the recommended way is to install some other most popular applications on the device and use them really, which can reveal integration issues that don't mesh well with the rest of the device.

Testers need to imitate real use scenarios to decrease the probability of problems found in real use. Testers must verify the common events of operating system interrupt, such as calls received, message arriving, device shutdown, etc. In addition, testers should be creative to produce certain types of events. For example, for a game, test low battery warning popping up when playing the game. Another example, for an application related to LBS (Location Based Service), step in an elevator without signal when using the application. In sum, the more you consider, the less potential problems will remain.

3) Server and service testing

Testers often focus on the client portion of the mobile application. In fact, most applications depend on a server or remote service to operate. If so, make sure thorough server and service testing is part of the overall test plan-not just the client portion implemented on the device.

Some fundamental tests, such as performance test and security test, should be covered for the application server. On this basis, QA testers should make special concern on the problems related to server upgrade, maintenance or service interruptions, because users always expect applications to be available any time. Testers should test if the users are notified when the service is unavailable and if the applications work well when the server is upgraded.

4) Related to charge, privacy and legacy

QA testers should test if an application complies with policies, protocols and agreements which the application must meet. These common agreements (if applicable) are Android License Agreement Requirements, Mobile Carrier/Operator Requirements (if applicable) and Application Certification Requirements, etc. There are some general and import rules in these agreements for QA concern: do not interfere with device phone and messaging services; do not break or exploit the device hardware and firmware; do not abuse or cause problems on operator networks.

Protection of private user data is always included in the above agreements. If your application accesses or uses private data, it is a good way to include an End User License Agreement [3] and a Privacy Policy with your application. Testers will check if this information is stored in plain text, and if it is transmitted without any safeguard.

If an application would cause the user to incur any fees, testers should test if the charge information is striking enough, if the delivery occurs when the user pays, otherwise the entire transaction is rolled back.

5) For good user experience

The first concern is installation and upgrade. QA testers should test installation on devices with low resources. If the application is available from the marketplace, you should test installation online or with the downloaded media. When a new version of the platform is released, you must re-test your application before your users are upgraded.

The second is user interface experience. QA testers may be check if screens is filled sparingly, if size graphics appropriately, and if the keys, clicks and glides are convenient.

The third is stability and responsiveness [8]. QA testers should test if the application start up fast and resume fast, if users are informed during long operations by using progress bars, and if resource consumption is reasonable.

Finally, do not forget to test features that are not readily apparent to the user, such as the backup/restore services, the sync features, and the help information.

D. Establishing a Mechanism for Usability Test

It is well known that good user experience is crucial for successful mobile applications. We just talked about test for good user experience, which is mainly a reference for your company's own QA testers to do some related tests. In this section, we introduce a mechanism for usability test which is different with the above mentioned. It is a combination of laboratory tests and field tests [9]. Laboratory tests are traditional way for usability tests, which are usually conducted in usability test laboratories, consisting of e.g. a living room or office-like area connected to a monitoring area with a one-way mirror [9]. While it is also concerned that laboratory evaluations do not simulate the context where mobile devices are used and lack the desired ecological

 ISBN: 978-1-61208-233-2

validity [9]. Interruptions, movement, noise, multitasking etc. that could affect the users' performance are not present in laboratory tests [9]. Therefore, field tests are worthwhile for mobile applications.

Both laboratory tests and field tests are through watching people actually use the application. They have several same key points in their procedures.

One is selecting of participated users. Generally, representative users are more likely to experience the same problems as the people who actually use the application [4]. So people who are representative of the target users conveniently, please do it. However, it isn't quite as important as it may seem, because many of the most serious usability problems are related to things like navigation, page layout, visual hierarchy, and so on problems that almost anybody will encounter [4]. So it's not always necessary and much more time-consuming and costly to find actual users. Whatever, it is very vital that the recruiter should be with reasonable common sense who's comfortable taking. We not only want to observe the user's action, but also want to know why to take the action.

Another is compiling test scenarios list. A good description should clarify the things you want them to try to do. Remember not to use research (unless search is being tested, of course) in the steps [4]. A pilot test of test scenarios should be done to find anything not clear in the scenario.

During the test, the observer should try to get the participants to externalize their thought process [4], and give neutral prompt to participants when encountering difficulties. Also, observers should be guaranteed to be able to observe the participant's action and words thoroughly.

Last, the debrief should take place as soon as possible after the test sessions, while what happened is still fresh in everyone's mind. Every observer can present their problems. These problems are summarized and arrayed by severity. Finally the top serious problems are chosen to be concerned primarily.

The greatest difference between laboratory tests and field tests is the context within which people uses the application. As a result, the time needed by field tests is more consuming. In general, when performing a user interface evaluation of mobile applications, laboratory tests can give sufficient information to improve the user interface and interaction of the system [9], not less than those found by field tests. While the field test method is suitable for situations where not only interaction with a system is tested, but also user behaviors and environment are examined [9]. In addition, confidentiality of the application or device in the industry often drives the decision towards the laboratory testing; especially in the beginning of the development cycle [9].

IV. CASE STUDY: TEST OF MOBILE APPLICATION OF QUESTIONNAIR SURVERY USING THIS FRAMEWORK

This section presents how the framework can be used in the mobile application of questionnaire survey.

We already have structured a devices database using MySQL and had some real devices in our test lab. This application was designed only for mobile phones of Android platform. Using the database information, QA testers choose two devices: MOTO XT800 with Android 2.0 and SAMSUNG I929 with Android 2.3. When designing the test plan, core function of the application are mainly filling the survey, submitting the survey, redeeming points and reviews; then additional test cases are designed because these two devices are customized by china telecomm; finally, several cases are designed for every device separately aiming at the differences introduced by different mobile OS version. According to a rough estimate, using the device system, we shorten the time for designing the whole test plan about 35%.

A defect system using software BugFree [10] has been built. The defects defined in Section III have been recorded in the defect tracking system. All team members can access the defect system. We also have received positive feedback about the convenience and the clarity by using the defect system.

We tested scenarios according to key points described in Section III. The application consumed a small quantity of local system resources, so no fatal problems were found about resources limitations. As for server testing, we found that if the service was closed unexpected and the client tried to connect the server, there was no obvious notification and the client kept waiting state. As to imitating real usage, an important question was found, that was while a survey was submitted, an incoming call failed. This phenomenon was obviously unreasonable. About charge and legal related field, the application is free; so, the test was simple and no problems were found. In user experience case, it was found that, in some survey, there were so many items that users had to turn pages for too many times if each page only for each item.

We invited six students to do usability test. As a final result, two reasonable advices were presented; first, the participants hoped to append progress bar or progress indicator in every page if the survey had many pages, so that the progress could be known at any time, and second, besides UC browser and Opera mobile web browser we used in the test, it was expected that QQ mobile browser [11], which is very popular in China, was adopted.

V. CONCLUSION AND FUTURE WORK

In this paper, a framework for QA testers to test mobile applications was proposed. This framework provides a helpful method to solve the question of heavy workload bought by mobile device's diversity and defines a specific defect system for mobile services to describe problems more accurately. Through our preliminary test practices, they are validated to be effective to shorten the time for designing test plan and preparation, and improve communication efficiency. Also, the framework suggests a set of test scenarios to be attended to particularly and highlights a mechanism for usability test. The benefit for product quality from them are proved in our test.

As future work, the framework should be applied in more testing of mobile applications. We should collect much more statistics to prove the benefit of the framework for mobile applications test. Also, we should refine and extend every part of the framework.

 ISBN: 978-1-61208-233-2

REFERENCES

[1] V. L. L. Dantas, F. G. Marinho, A. L. da Costa, and R. M. C. Andrade, "Testing Requirements for Mobile Applications", Computer and Information Sciences, 2009. ISCIS 2009. 24th International Symposium on, Sept. 2009, pp. 555-560, doi:10.1109/ISCIS.2009.5291880.

[2] D. Franke and C. Weise, "Providing a Software Quality Framework for Testing of Mobile Applications", 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, Mar. 2011, pp. 431-434, doi: 10.1109/ICST.2011.18.

[3] S. Conder and L. Darcey, "Android Wireless Application Development (2nd edition)", Addison-Wesley, 2011.

[4] S. Krug ,"Rocket surgery made easy", New Riders, 2010.

[5] N. S. Godbole, "software quality assurance principles and practice", Alpha Science Intl Ltd, 2004.

[6] C. Woody, N. Mead and D. Shoemaker, "Foundations for Software Assurance", 2012 45th Hawaii International Conference on System Sciences, Jan. 2012, pp. 5368-5374, doi: 10.1109/HICSS.2012.287.

[7] V. Roto, "Web Browsing on Mobile Phones-Characteristics of User Experience", Doctoral Dissertation, TKK Dissertations 49, Espoo 2006.

[8] R. Hoekman Jr., "Designing the Obvious: A Common Sense Approach to Web & Mobile Application Design (2nd Edition)", New Riders Press, 2010.

[9] A. Kaikkonen, T. Kallio, A. Kekäläinen, A. Kankainen, and M. Cankar, "Usability Testing of Mobile Applications: A Comparison between Laboratory and Field Testing", Journal of Usability Studies, Issue 1, Vol. 1, Nov. 2005, pp. 4-16.

[10] http://www.bugfree.org.cn/ [retrieved: September, 2012].

[11] http://mb.qq.com/ [retrieved: September, 2012].

 ISBN: 978-1-61208-233-2

Variability Management in Testing Architectures for Embedded Control Systems

Goiuria Sagardui, Leire Etxeberria and Joseba A. Agirre
Computer and Electronics department
Mondragon Goi Eskola Politeknikoa
Loramendi 4, Mondragón (Gipuzkoa), Spain
Email: {gsagardui, letxeberria, jaagirre}@mondragon.edu

***Abstract*— In recent years, embedded systems have substantially increased their presence both in industry and in our everyday lives. Hence, more and more effort is being dedicated to the development of such systems. Since embedded systems involve computation that is subject to physical constraints, the development and validation of software for such systems becomes a challenge. Moreover, the validation of the embedded system within the environment increases the complexity and cost of testing, so many efforts are being devoted to perform testing activities from early phases of the development. Testing by simulation of the system and its environment is one of the most promising approaches to reduce testing costs. In this paper, we present a proposal based on model-based testing and variability management and integrated in Simulink for ensuring the correctness of a embedded control software. Variability management of configurations helps managing different simulation environments and allows less costly and time-consuming testing.**

Keywords - testing architecture; variability management; simulation.

I. INTRODUCTION

Embedded systems are engineering artifacts involving computation that is subject to physical constraints. The physical constraints arise through two kinds of interactions of computational processes with the physical world: (i) reaction to a physical environment, and (ii) execution on a physical platform [1]. Concentrating on software, embedded system software characterizes itself, among others, by heterogeneity, distribution (on potential multiple and heterogeneous hardware resources), ability to react (supervision, user interfaces modes), criticality, real-time and consumption constraints [2]. The need to consider all these factors in concert makes the development of software for embedded systems a complex endeavour.

However, not only development poses a significant challenge. Due to its complexity, the validation of embedded software also becomes a cumbersome task. Embedded software needs to cater for the variability on both the physical environment and the physical platform it is executed on apart from testing the software itself.

Moreover, when we consider that embedded systems are often part of safety-critical systems (e.g., aviation or railway systems), the validation of the software becomes essential [3], which also raises testing cost.

Model Driven Engineering (MDE) is a paradigm that promises a reduction in testing efforts. Models become the central asset of the development so testing can be started from early phases. Model-, software-, processor-, and hardware-in-the-loop (MiL, SiL, PiL, and HiL) tests; called X-in-the-loop tests provide four testing configurations [4]. "The model, software, processor, and hardware terms refer to the different target system configurations in the testing environment, each of which adds value to the verification process" [4].

The MiL tests the model along with the plant model that simulates the physical environment signals. For SiL testing, the model of the MiL is replaced with the corresponding software code. This source code can be autogenerated from the model. PiL tests the source code executed on the target processor machine. For HiL testing, the software is integrated with the real software infrastructure and deployed in the hardware processor or microcontroller. The environment around the system is still a simulated one, but the plan model is replaced by a dedicated hardware setup specially designed for the simulation [4].

Each of the configurations has a different focus from the validation point of view and following them allows detecting errors early when they are easier to correct and to validate incrementally different aspects of the system (functionality, performance, etc.). Functionality and system behavior can be tested at MiL and SiL level. Tests on PiL level can reveal faults that are caused by the target compiler or by the processor architecture [5]. HiL level is to reveal faults in the low-level services and in the I/O services [5]; and to confirm the real-time functionality and performance [4].

Embedded software for control systems usually has to run in different environment conditions, and has to control different number or/and types of sensors and actuators. This increases the complexity of testing even in early phases of the development. Testing the control system in different real scenarios is very costly and time-consuming.

Taking into account variability in different aspects of the validation from early testing architectures allows reducing the testing complexity by considering all the possible variants both in software, tests and the environment from simulation. This ensures an increased coverage of the testing in early phases of the development and a correct selection of the most risky scenarios for testing the final system.

This paper proposes a systematic approach to X-in-the-Loop validation considering the variability in the testing

 ISBN: 978-1-61208-233-2

architecture. The proposed variability management can be reused along the testing process (MiL, SiL and PiL).

Simulink [6] was chosen as the simulation framework to simulate the real environment in which software should be integrated.

The paper is structured in the following way: Section II presents the background and the state of the art, Section III discusses about variability in testing architecture, Section IV presents the variable simulation model and, to finish, conclusion and future work are stated in Section V.

II. BACKGROUND

This section provides a brief introduction to the background.

A. Embedded Systems Engineering

The function of Systems Engineering is to guide the development of complex systems, understanding system as a set of interrelated components working together toward some common objective [7]. Embedded systems are a particular type of system, where the system is embedded in its enclosing device (e.g., elevators). There is an essential difference between embedded and other computing systems that makes their engineering particularly challenging. Since embedded systems involve computation that is subject to physical constraints, the separation of computation (software) from physicality (platform and environment) does not work for embedded systems. Instead, the design of embedded systems requires a holistic approach that integrates hardware design, software design, and control theory in a consistent manner [1].

B. Model-based System Engineering

"Model-based Systems Engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases" [8]. "MBSE is part of a long-term trend toward model-centric approaches adopted by other engineering disciplines, including mechanical, electrical and software" [8]. In the particular case of software, MBSE can be seen as part of Model Driven Engineering (MDE), a software development paradigm where models are the central element in the development process [9]. Hence, following MDE, systems software does not only serve as documentation, but can also be used to generate code or be executed for validation purposes.

C. Variability Management

Variability is the ability to change or customize a system [10]. Variability can also be understood as modifiability (to allow variation or evolution over time) and configurability (variability in the product space) to get a set of related products or different configurations [11]. Variability and its management are key aspects not only in software product lines, but in other systems such as embedded systems. Many variability modeling techniques have been developed. Several of the approaches are based on feature modeling, one of the most used technique for variability modeling: [12][13][14], etc. There are other approaches that are based on use cases [15] or approaches that use both feature models and use cases such as [16] and [17]. Other approaches model variation points such as [18][19] and [20]. There are also approaches that integrate variability in ADLs (Architecture Description Languages) such as Koalish [21] and [22]. Several techniques use UML (Unified Modelling Language) that is the de facto notation standard in industry for software modelling. UML profiles or extensions to UML are proposed to introduce variability [23][24][25], etc.

[26] presents an approach for managing variability in Simulink models using Pure:variants for Simulink. Another approach that addresses variability in Simulink models is the approach for model-based embedded software product lines of [27][28][29]. [30] also addresses variability in Simulink.

All these approaches address variability in Simulink models. However, the focus is on managing the variability in model-based embedded systems and product lines.

Our approach is more oriented towards testing and how to manage variability in a test architecture; a Simulink model is used for implementing a test architecture. In addition to the variability in the simulation environment represented in the Simulink model, variability in the Software under test and test specifications is also considered.

Regarding variability management during validation, [31] defines a software product line for validation environments, to support variability in those environments and to be able to test different applications in different domains and technologies.

III. VARIABILITY IN TESTING ARCHITECTURES

Testing architectures are the base for a systematic testing process. By defining the testing architecture from initial phases, we ensure a correct definition of the tests and a reutilization of them along the lifecycle.

We have defined the testing architecture in Simulink [6]. Simulink is a commercial tool for modeling, simulating and analyzing multidomain dynamic systems that is integrated into the MatLab programming environment. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries [6]. Simulink block diagrams define time-based relationships between signals and state variables. Signals represent quantities that change over time and are defined for all points in time between the block diagram's start and stop time. The relationships between signals (input and output) and state variables are defined by a set of equations represented by blocks [32].

A testing architecture can be structured in four key elements; each element and the implementation of those elements using a Simulink model is explained:

-*Sources*: the inputs are the test cases to execute on the system under test. A test case is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not. In Simulink, test cases will define the set of signals that determine the simulation of the environment in which the system has to run (plant). Usually a mixture of both signals and plants ensures a correct X-In-The-Loop simulation

 ISBN: 978-1-61208-233-2

-*System-Under-Test (SUT):* at early phases of the development, the SUT is a model of the system. Then the code of the system can be simulated (S-Function in Simulink) and in final stages of development, the code can be tested within the running platform. The software can be developed following an Software Product Line (SPL) methodology or as a single system. A Software Product Line is a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [33]. In this type of development, variability of the software is instantiated at design time so the final software will have the functions for the concrete configuration in which the software will run. When the development follows a single system development, it usual to have configurable software. In this case, the software has all the functions in all configurations, but by defining the values of some parameters, the software will execute as expected in each configuration.

-*Metrics*: the metrics automatically analyze the test results for each test case. In Simulink, one can use verification blocks associated with the output signals to decide automatically on the correctness of the results of the test.

-Test Control: In Simulink, it is a block that controls the order of the test cases.

A. *Case study: Door management control*

The proposed approach has been applied in a door management control system of an elevator. This system controls the opening and closing of the doors (that include sensors, motors, etc.).

The behavior of the control is specified using a state machine where the states (Idle, Open, Opening, Closing, Closed) and actions to be applied in each state are defined. This state machine has been specified using Iar Visual State tool [34] and the code has been automatically generated. This code has been introduced in the Simulink model that implements the testing architecture as a block (an S-function, a computer language description of a Simulink block).

In Figure 1, the testing for the door management control is described:

- Test sequences indicate the values of signals over time. These sequences are automatically generated from abstract behaviour models of the software that are annotated with time aspects.
- Software-Under-Test is automatically generated from models in Iar Visual State and transformed to SFunction for integrating in Simulink.
- Model is simulated and results (output signals over the time) are obtained. These results are used to compare with expected ones.

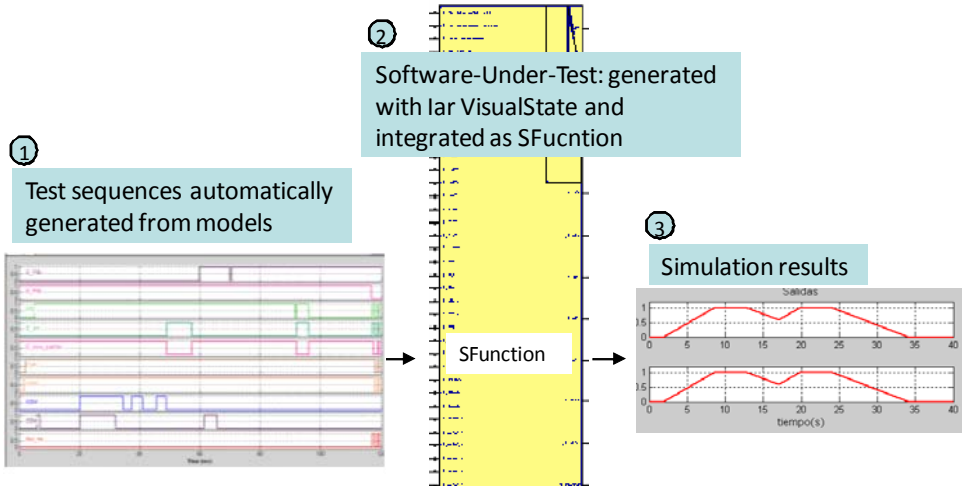


Figure 1. Simulation Environment

The simulation environment (the plant model that simulated the physical environment signals) is valid only for one concrete configuration. One of the factors that add more complexity to testing of embedded software is the diversity of environments in which software can execute. Embedded software usually execute under different configurations. It can be connected to different number of devices, etc. There is a need to manage the variability in validation environment due to: number and type of sensors, number and type of actuators, communication mechanisms, etc.

In order to identify and model the environments in which software should be validated, a feature model can be used. A feature model is an and/or tree of different features. A feature as “a prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems” [12]. Features can be mandatory, optional or alternative. Features are an effective way of identifying the variability (and the commonality) among different products in a domain. Moreover, features are a effective means of communication among stakeholders and are a intuitive way of expressing the variability [35] as features are distinctive

 ISBN: 978-1-61208-233-2

characteristics or properties of a product that differ from others or from earlier versions.

The feature model contains the different elements that should be considered when validating the software depending on sensors, actuators, etc. of each configuration. Some of the most relevant are:

- Different types of doors: Software must be validated with different types of Doors: articulated, non-articulated, etc.
- Different number of doors: Software must be validated with one, two, three Doors. Doors can operate independently or not.
- Different floor configuration: floors can have different door configuration: depending on the floor, doors require different behaviour.
- Different sensors: Optional obstacle and presence sensor and optional limit switch.

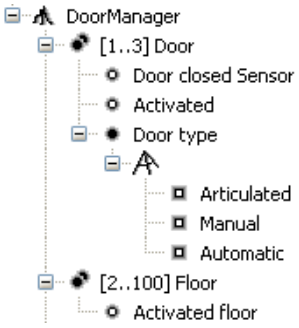


Figure 2. Feature model for validation

Feature model containing the variability can be modelled as a form in MatLab or using some specific tool for variability management, such as pure::variants for Simulink. To perform validation taking into account this variability, it is necessary to manage the variability in the following aspects of the simulation environment:

- Configuration of the Software-Under-Test. A .xml file is automatically generated from variability management form and indicates the initialisation of the SUT for a concrete configuration.
- Modelling of the simulation environment: In Simulink, variability is on relations and blocks that are required for simulation. Simulation elements are contained in a library and are connected automatically guided by the variability form in order to create the simulation model for a configuration.
- Tests' specification: As not all the configurations require the same requirements for testing, variability in tests should be taken into account too. Depending on the configuration some functionalities are not active or even, same functionalities could differ on required response time.

The next section details this variable simulation model.

IV. VARIABLE SIMULATION MODEL

Simulation model includes the simulation of both mechanicals elements and software that manages these elements. Including variability in the simulation models allows representing different configurations in which software will run. This way it is possible to validate the system taking into account different configurations in a less costly and time-consuming way. The software is integrated as a block in the model and is connected to the blocks representing the mechanicals elements. Running the model provides the simulation of the real system.

In order to get an effective simulation of different configurations, variability management has been included in Simulink. For this purpose, a variability form has been developed in MatLab asking for the information that represents the configurations: number and types of doors, etc.

This information is used to develop dynamically the simulation model of the configuration to be validated. In order to develop the simulation model dynamically we have created a library with the elements that can appear in the simulation model: doors, code block, etc. Code is executed for creating the simulation model with the features selected in the variability form. See Figure 3 for simulation model creation for a configuration containing two doors of NormalType.

```
function CreateNormalDoor(n)

open_system('elevatorLibrary');
open_system('installation');
for i = 1:n
      add_block('elevatorLibrary/door', 'installation/door', 'MakeNameUnique', 'on');
      add_line('installation','code/1','door/1');
end

end
```

Figure 3. MatLab code for dynamic simulation model creation

This way, by selecting values in the variability form, we obtain automatically simulation models that are specific for the configuration we want to prove (See Figure 4). The same test architecture is used and test cases may be also adapted and reused as test cases will be also developed taking into account variability. Thus, we obtain the advantage of getting simulation models for different configurations with a reduced cost. Therefore, tests could be easily performed in different configurations obtaining greater test coverage of the embedded system. In an initial development stage, the simulation models may be automatically generated in an exhaustive way to test all configurations. In later stages and during maintenance, the generation of simulation models may be used to test new configurations.

 ISBN: 978-1-61208-233-2

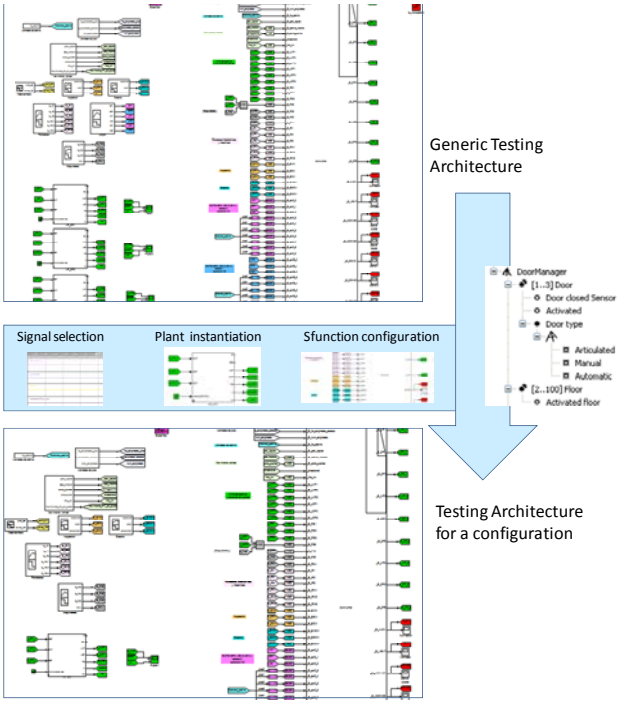


Figure 4. Instantiation of the testing architecture for an configuration

V. CONCLUSION AND FUTURE WORK

This paper has described a simulation environment for embedded software based on model based testing and variability management and using Simulink as simulation tool. As embedded software usually runs under different configurations, it is costly to test the software under real conditions. Variability management of configurations helps automating the simulation environment. This way, validation is simplified and intensive testing can be performed.

In the case study, a variability management form has been developed in MatLab. This option has been adequate for our purpose, but as complexity increases it is recommended to use a tool specific for variability management. Pure::Variants is a tool integrated with Simulink that could be adequate for this purpose [26]. Or, the variability management approach for Simulink proposed by [27][28][29] can be also used for plant instantiation part. Those approaches can be used in a complementary way for variability management and instantiation of the Simulink models (plant). Those approaches are not oriented to manage variability in validation architectures, but in general in Simulink models, so they do not cover some specific needs such as the configuration of Sfunctions in Simulink, generation of test sequences, etc., that our approach covers.

It is always difficult to establish the coverage of the tests, more when multiple configurations have to be validated. Although tests cover 100% of transitions we can not ensure that all configurations have been tested. In this case, variability instantiation has been done manually. In order to get a greater coverage, it is highly recommended to automate the variability selection, generating the simulation environment for all the configurations sequentially. Next steps include analysing the feasibility of this option and the coverage that is got this way.

The paper has focused on the simulation phase. However, once a system has been validated in Simulink, software is integrated in the real system. Test architecture and variability management should be reused in subsequent phases. Our next actions will consider the generation of tests from the model for running in the software using python test scripts [36].

ACKNOWLEDGMENT

This work is co-supported by the Basque Government under grants UE2011-4 (COMODE Project). The project has been developed by the embedded system group supported by the Department of Education, Universities and Research of the Basque Government.

REFERENCES

[1] T. A. Henzinger and J. Sifakis. “The Embedded Systems Design Challenge,” In 14th International Symposium on Formal Methods (FM 2006), Hamilton, Canada, volume 4085 of Lecture Notes in Computer Science, pp. 1–15. Springer, 2006.

[2] OMG. “UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems”. Formal Specification, November 2009. Online at: http://www.omg.org/spec/MARTE/1.0/PDF. [retrieved: September, 2012].

[3] J. A. Stankovic. “Strategic Directions in Real-time and Embedded Systems”. ACM Computing Surveys, 28, pp. 751–763, December 1996.

[4] H. Shokry and M. Hinchey. “Model-Based Verification of Embedded Software”, IEEE Computer, vol. 42, no. 4, pp. 53-59, April, 2009

[5] E. Bringmann and A. Krämer. “Model-based Testing of Automotive Systems” pp. 485–493, 2008 International Conference on Software Testing, Verification and Validation, ICST, 2008.

[6] Simulink webpage. http://www.mathworks.com/products/simulink/. [retrieved: September, 2012].

[7] A. Kossiakoff and W. N. Sweet. Systems Engineering. Principles and Practice. Addison Wesley, 2003.

[8] International Council on Systems Engineering (INCOSE). “Systems Engineering Vision 2020”. Technical Report INCOSE-TP-2004-004-02, INCOSE, September 2007.

[9] R. France and B. Rumpe. “Model-Driven Development of Complex Software: A Research Roadmap”. In Workshop on the Future of Software Engineering (FOSE 2007), at ICSE 2007, Minneapolis, Minnesota, USA, pp. 37–54, 2007.

[10] J. Van Gurp, J.Bosch, and M. Svahnberg. “On the notion of variability in software product lines”. In WICSA ’01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA’01), Washington, DC, USA, 2001. pp. 45-54. IEEE Computer Society.

[11] S. Thiel and A. Hein. “Systematic integration of variability into product line architecture design”. In SPLC 2: Proceedings of the Second International Conference on Software Product Lines, pp. 130–153, London, UK, 2002. Springer-Verlag.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. “Feature-oriented domain analysis (foda) feasibility study”. Technical Report CMU/SEI-90-TR-21, November 1990.

[13] K. Czarnecki and U. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, June 2000.

[14] K. Czarnecki, S. Helsen, and U. W. Eisenecker. “Staged configuration using feature models”. In Robert L. Nord, editor, SPLC,

 ISBN: 978-1-61208-233-2

volume 3154 of Lecture Notes in Computer Science, pp. 266–283. Springer, 2004.

[15] G. Halmans and K. Pohl. “Communicating the variability of a software-product family to customers”. Software and System Modeling, 2(1): pp. 15–36, 2003

[16] M. Eriksson, J. Börstler, and K. Borg. “The pluss approach - domain modeling with features, use cases and use case realizations”. In J. Henk Obbink and Klaus Pohl, editors, SPLC, volume 3714 of Lecture Notes in Computer Science, pp. 33–44. Springer, 2005.

[17] T. von der Maßen and H. Lichter, “Requiline: A requirements engineering tool for software product lines”. In 5th International Workshop on Product Family Engineering, PFE5, Proceedings, 2003, pp. 168-180.

[18] H. Gomaa and D. L. Webber. “Modeling adaptive and evolvable software product lines using the variation point model”. In HICSS ’04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS’04) - Track 9, p. 90268.3, Washington, DC, USA, 2004. IEEE Computer Society.

[19] K. Pohl, G. Böckle, and F. J. van der Linden. Software Product Line Engineering : Foundations, Principles and Techniques. Springer, September 2005.

[20] M. Becker. “Towards a general model of variability in product families”. In Proceedings of the 1st Workshop on Software Variability Management, 2003.

[21] T. Asikainen, T. Soininen, and T. Männistö. “A koala-based approach for modelling and deploying configurable software product families”. In Frank van der Linden, editor, Software Product-Family Engineering, 5th International Workshop, PFE, Revised Papers, volume 3014 of Lecture Notes in Computer Science, pp. 225–249. Springer, 2003.

[22] A. van der Hoek. “Design-time product line architectures for any-time variability”. Sci. Comput. Program., 53(3), pp.285–304, 2004.

[23] H. Gomaa. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley, 2004.

[24] M.s Clauß. “Modeling variability with uml”. In Proceedings of GCSE2001. Young Researchers Workshop, 2001.

[25] T. Ziadi, L. Hélouët, and J.M.c Jézéquel. “Towards a uml profile for software product lines”. In Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, Revised Papers, pp. 129–139, 2003.

[26] C. Dziobek, J. Loew, W. Przystas, and J. Weiland. “Model diversity and variability - handling of functional variants in simulink-models”. Elektronik automotive, February 2008, pp.33-37.

[27] A. Polzer, D. Merschen, A. Botterweck, G. Pleuss, J. Thomas, S. Hedenetz, and B. Kowalewski. “Managing complexity and variability of a model-based embedded software product line”. Innovations in Systems and Software Engineering (ISSE), 8, pp.35–49, 2011.

[28] G. Botterweck, A. Polzer, and S. Kowalewski. “Using higher-order transformations to derive variability mechanism for embedded systems”. 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACESMB 2009) atMoDELS 2009, Vol-507, pp. 107 – 121, Denver, Colorado, USA, September 2009.

[29] G. Botterweck, A.s Polzer, and S Kowalewski. “Variability and evolution in model-based engineering of embedded systems”. In 6. Dagstuhl-Workshop Model-Based Development of EmbeddedSystems (MBEES 2010), pp. 87–96, Dagstuhl, Germany, February 2010.

[30] D. Beuche and J. Weiland. “Managing flexibility: Modeling binding-times in simulink”. In Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA ’09, pp. 289–300, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] B. Magro, J. Garbajosa, and J. Pérez. “A software product line definition for validation environments”. In Proceedings of the 2008 12th International Software Product Line Conference, SPLC ’08, pp. 45–54, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Modeling Dynamic Systems, web page, http://www.mathworks.es/es/help/simulink/ug/modeling-dynamic-systems.html. [retrieved: September, 2012].

[33] P. Clements and L. Northrop. Software Product Lines - Practices and Patterns. Addison-Wesley, 2001.

[34] IAR VisualState, Web page. http://www.iar.com/en/Products/IAR-visualSTATE/, [retrieved: September, 2012].

[35] K. Lee, K. C. Kang, and J. Lee. “Concepts and guidelines of feature modeling for product line software engineering”. In ICSR-7: Proceedings of the 7th International Conference on Software Reuse, pp. 62–77, London, UK, 2002. Springer-Verlag.

[36] Python official webpage. http://www.python.org/. [retrieved: September, 2012].

 ISBN: 978-1-61208-233-2

GUI Failure Analysis and Classification for the Development of In-Vehicle Infotainment

Daniel Mauser
Daimler AG
Ulm, Germany
daniel.mauser@daimler.com

Alexander Klaus
Fraunhofer IESE
Kaiserslautern, Germany
alexander.klaus@iese.fraunhofer.de

Ran Zhang
Robert Bosch GmbH
Leonberg, Germany
ran.zhang@de.bosch.com

Linshu Duan
AUDI AG
Ingolstadt, Germany
linshu.duan@audi.de

***Abstract*—Modern automotive infotainment systems have sophisticated graphical user interfaces, leading to various challenges in software testing. Due to the enormous amount of possible interactions, test engineers have to decide which test aspects to focus on. In this paper, we examine what types of failures can be found in graphical user interfaces for automotive infotainment systems and how frequently they occur. A hierarchical classification for failures has been developed based on common concepts in software engineering, such as Model-View-Controller and Screens. More than 3,000 failures, found and fixed during the development of automotive infotainment systems at Audi, Bosch and Mercedes-Benz, have been analyzed. Results show that 62% of reports describe failures related to high and low level behavior, 25% of reports describe failures related to contents and 6% of reports describe failures related to design.**

***Keywords*-failure reports; domain specific failures; GUI based software; in-vehicle infotainment system.**

I. INTRODUCTION

In modern automotive infotainment systems the graphical user interface (GUI) is an essential part of the software. The so-called HMIs (human machine interface) provide the system functionality to the user, whether it be the radio system, the navigation, or system functionality, such as the tire pressure monitoring system. According to Robinson and Brooks [1], a GUI "is essential to customers, who must use it whenever they need to interact with the system". Additionally, they "found that the majority of customer-reported GUI defects had major impact on their day-to-day operations, but were not fixed until the next major release" [1]. As automotive infotainment GUIs are built into a car, there is no easy possibility to upgrade the system or to buy a new release, which renders the situation for such systems even worse. Additionally, when the system does not work correctly, drivers may get distracted from driving. Therefore, special attention has to be drawn on finding and fixing defects during development.

Figure 1 shows an example of a screen in an HMI. It consists of a menu at the top of the screen, where all available applications, e.g., navigation or audio, can be accessed. Each application consists of an application area at the middle of the screen, where the actual content is displayed (here: information about the radio station and the song played) and a sub menu for content specific options at the bottom (here: "Radio", "Presets", "Info", etc.). The HMI is operated via a central control element (CCE) allowing the user to set the selection focus by rotating or pushing the CCE in a direction, and to activate options by pressing it down. This interaction concept is common in modern in-vehicle infotainment systems.

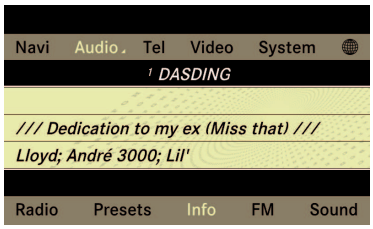


Figure 1. Example for a graphical user interface of the Mercedes-Benz infotainment system COMAND

GUI based software, especially in the automotive domain, is becoming more and more complicated - often, documents with more than 2,000 pages are written to describe all the functionality [2]. The reasons are the growing amount of functions, which form more and more complex systems, as well as the usage of more advanced graphical views and elements (e.g., complicated animations or 3D-elements).

When testing GUIs, sequences of system interactions are performed and the system reaction is compared to the specified reaction in each step. It is obvious that for such complicated systems, not all possible combinations can be tested, and thus it is necessary to focus testing activities on certain failure types. To be able to choose strategies accordingly, several questions need to be answered:

- What types of failures can be found in GUI based software today? Is it possible to build a classification of these types?
- What are frequent failures in current GUI software? Which are common, which are rare?

The article is structured as follows. In Section 2, we discuss related work and show why we need to create a new classification scheme. Section 3 describes our approach

 ISBN: 978-1-61208-233-2

for creating a scheme, which is then presented in Section 4. Section 5 discusses the results of our work. The article ends with a discussion of the approach and future work in Section 6.

II. Related Work

In the literature, various types of defect classifications can be found. However, many of them lack the practical usage and empirical data in the form of distributions of defects into the scheme, and thus it is hard to tell whether they are a valuable addition. Other schemes for classification are used frequently, or at least once. For our study, we concentrate on those latter ones, and discuss why they are not fully suited for our means. As described above, our context is black-box testing of a GUI for automotive infotainment systems.

IBM created the so-called Orthogonal Defect Classification (ODC) [3]. Since then, many companies have applied this approach. It consists of several attributes, such as triggers, defect types, impact, and others. A GUI-section is included in the ODC Extensions V5.11. It contains triggers, such as "Design Conformance", "Navigation", and "Widget/GUI Behavior". Compared to our classification (see Section 4), some of the triggers match to our categories, but not fully: "Navigation" is represented by "Screen Transition" and "Widget Behavior", which also maps to the "Widget Behavior" trigger. Our "Design" category maps to "Icon Appearance" and "Design Conformance", while "Input Devices" are not covered by us. All in all, the scheme would be spread across three levels of our classification.

Another scheme, which contains several categories for GUI-related issues, was made by Li et al. [4]. It consists of 300 categories, and is based on the ODC, but adapted for black-box testing. It contains, e.g., categories for GUIs in general, and for GUI control [4]. The GUI-related categories do not fully fit, for example, there is a "Title bar" category, but our systems do not have title bars, as desktop software does. This scheme is created for regular desktop software, as it also classifies keyboard or mouse related faults. Due to the differences between desktop software and our systems, we decided not to adapt this scheme.

Børretzen and Dyre-Hansen [5] created a scheme, which is also based on the ODC. They target industrial projects. A GUI fault category is included, but not further segmented. The rationale for this is that, although "function and GUI faults are the most common fault types", they are most often not severe, and thus, not as critical as other categories [5]. This seems to be a contradiction to what was stated in the introduction, but the criticalities of certain types of faults are subject to the application domain. As stated in the introduction, in our application domain they are very critical, and therefore, we focus on them to assure software quality.

Hewlett-Packard created a scheme based on three categories: origin, type, and mode [6]. Origin refers to where the defect was introduced, the type can, e.g., be logic, computation, or user interface. The mode refers to whether something was missing, unclear, wrong, changed, or done in a better way now. This scheme also does not differentiate the various types of GUI-related failures.

Another well-known scheme has been developed by Beizer [7]. The main categories are "requirements, features and functionality, structure, data, implementation and coding, integration, system and software architecture, and testing" [7, p. 33], each having three levels of subcategories. The scheme is very detailed, but there is no GUI-related category.

An adaption of this scheme for GUI contexts has been created by Brooks, Robinson and Memon [8]. The authors emphasize that "defining a GUI-fault classification scheme remains an open area for research" [8]. They simplified Beizer's scheme to create a two level classification and added a subcategory for GUI-related issues, "to categorize defects that exist either in the graphical elements of the GUI or in the interaction between the GUI and the underlying application" [8]. However, all of our failures would fit into that category, and thus, we cannot use this scheme.

There also exists a fault classification scheme for automotive infotainment systems [9], however, this scheme is based on the network communication, and thus, it cannot be used for our purposes of classifying software based GUI failures. This scheme differs between hardware and software, but does not differentiate the different possibilities of issues in the software enough: "software based system faults can be computational, data management and interface faults" [9]. This scheme again has many categories not usable by us, and does not include different GUI-related categories.

Ploski et al. [10] studied several schemes for classification, including approaches not presented here. Since there was no matching scheme, we did not present them here.

Another approach has been created by the IEEE [11]. However, this approach is not very detailed, and just lists a number of attributes to be filled out for each defect. But the standard includes a scheme for distinguishing between defects and failures. A defect is "an imperfection or deficiency in a work product that does not meet its requirements or specifications", while a failure is "an event, in which a system or system component does not perform a required function within specified limits" [11]. So, when a defect is present, and we perform GUI testing, we can observe failures. They are caused by defects in the code, but since we test by using the GUI, and not the code (i.e., black box), what we can observe is the behavior, and this is why we do not create a defect but a failure classification scheme.

The classification schemes available do not meet our requirements. Since we employ block-box testing of GUIs, we cannot use any code-related categories or schemes. We focus only on GUI-related failures. The schemes presented in [6][7] and [9] do not have GUI-related categories and because of this, they cannot be used by us. Others ([3][5][8])

 ISBN: 978-1-61208-233-2

have GUI-related categories, but still do not meet very well to our purposes. The scheme presented in [4] has many GUI-related categories, but for desktop software. Due to the differences of desktop and automotive infotainment GUIs, we did not adapt it because we would then have to either delete or change most of the categories. Therefore, we created our own failure classification scheme. After describing the approach we used, the categories of our scheme are explained in Section 4.

III. METHODOLOGY

For this research, we analyzed databases of existing failure reports. The data was collected during the development of state of the art automotive infotainment systems. Testers executed the System Under Test (SUT) manually, based on specification documents, and used failure reporting tools to keep records of anomalies. The reports were handed over to developers who then rechecked and fixed the software. In this context, failures are defined as mismatch between the SUT and an explicit GUI specification, which can be observed while operating the system. Any implicit requirements, such as general standards or guidelines, are not subject of the study. Only reports that were accepted as failures by both testers and developers were accounted. Failures that are not referring to the GUI were sorted out.

For this study, Audi, Bosch and Mercedes-Benz provided failure data. Hence the analyzed reports represent a broad variety of contexts as they stand for different infotainment systems (Audi MMI, Mercedes-Benz COMAND and several projects, developed at Bosch), different steps in the development process as well as different test strategies, test personnel and test environments. In total, more than 3,000 reports were analyzed. One third of the reports have been used as training data to construct the failure classification which then was fine-tuned using the remaining reports as test data.

As preparation of the analysis, the reports were exported to an Excel document with one line for each report. Furthermore, reports that describe more than one failure have been split up in one line for each failure. Redundant reports that describe exactly the same failure as already considered ones were removed. The following information per report was relevant for the analysis:

A **Report ID** identifies the reports uniquely. In the **Title** testers describe the essence of the report. The **Problem description** is a detailed statement on (a) the required setup of the system under test, (b) the actions that lead to the failure, (c) the behavior or result that has been observed, (d) a description, what should have been displayed instead and (e) how this failure could be bypassed. If failures were ambiguous or hard to describe, screen shots were added. Table I shows simple examples of reports.

We analyzed this data iteratively by hand to develop a classification by clustering similar failures. To determine the similarity of failures, the classification is based on concepts and patterns used in software engineering. For example, the top level failure classes are *behavior*, *contents*, and *design*, according to the well-established Model-View-Controller [12] design pattern. The structure of the classification and related separation criteria are presented in Section 4.

Table I
EXAMPLES OF THE ANALYZED GIVEN FAILURE REPORTS

ID	Title	Problem description
4711	Inserted music CDs are not played automatically	*Setup*: Any state *Actions*: Insert music CD *Observed result*: Nothing happens *Expected result*: System should display CD play screen *Reference*: R0026679 *Workaround*: Navigate to CD play screen manually
4712	Cell phone icon on call screen obsolete	*Setup*: Connect cell phone *Actions*: Navigate to Call screen *Observed result*: Placeholder icon for cell phones is displayed *Expected result*: Correct icon is displayed *Reference*: R0026672 *Workaround*: —

A classification is needed that gives a good overview and is flexible to extend for comprehensiveness. This should be achieved by a hierarchical structure. As indication, how many hierarchy levels have to be applied and whether one category could be subdivided reasonably or several categories should be combined, we defined the following requirements for the failure classes: To scale the scope of each classification level, an initial analysis of the data indicates the necessity to limit the percentage of the lowest level to 10% of the total numbers of failures. To develop a clear and easy to use structure, the number of categories on every level has to be 2 in minimum and 5 in maximum.

IV. FAILURE CLASSIFICATION

In this section, the GUI failure report classification is described. Table II gives an overview of the entire classification, including the failure distribution. As mentioned above, the top level follows the Model-View-Controller pattern [12], as this pattern proved to be an adequate abstraction for GUI based software. **Controllers** (here: *behavior*) abstract the observable behavior, indicating how input is processed. **Models** (here: *contents*) define all contents that are displayed by the system. **Views** (here: *design*) describe layout and appearance of the contents to be displayed. As the SUT was tested as a black box, the MVC pattern is not intended to represent the actual software structure or to relate any failures to implemented software modules.

In order to avoid enforced classifications of reports to existing classes, one category “to be categorized” (TBC) has been created. As for other categories, on the lowest level the TBC failure class is limited to 10% of the total number of

 ISBN: 978-1-61208-233-2

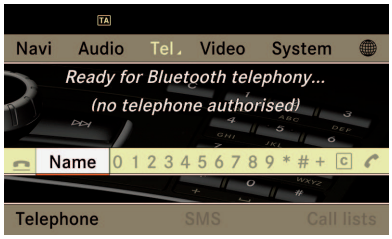


Figure 2. Screen example: Telephone application

failures. Classifying more failures than that limit as TBC would indicate, that the definition of an additional failure class is necessary.

A. Behavior

The top level failure class *behavior* contains all failure reports describing that stimuli to the SUT do not result in the specified output. In order to subdivide this failure class, common abstractions in GUI development were applied:

Screens [13][14] represent the current state of the GUI displayed. This state is defined by the options available to the user. Figure 1 shows the radio screen, where the current radio station and the song playing are displayed. The options provided allow users to change waveband (FM option) or adjust the sound setting (Sound option). Screens are structured based on elementary GUI elements, so called **widgets**. Widgets are either primitive (label, rectangle, etc.) or complex, meaning that they are a composition of primitive or again complex widgets. An example for widgets in Figure 1 would be the horizontal list in the top end that contains button widgets for all available applications, such as “Navi”, “Audio” or “Tel” (i.e., phone). In this classification, the concepts of screens and widgets are used to differentiate micro behavior that affects single elements on the display (e.g., iterating list entries) and macro behavior that changes the entire context of use.

1) Widget Failures: The GUIs of the automotive infotainment systems analyzed mainly use various types of lists to present options to the user. To activate an option, those lists set a focus by turning or pushing the CCE and pressing the CCE once the option wanted is focused. Potential failures might be that the wrong option is focused on start or that the focus changes not as specified. An example would be that every time the main menu is entered, the element in the middle should be focused automatically. A failure would exist, if the first element would be focused instead. Those failures are considered as deficient *widgets focus* logic. Subcategories are *initial focus* (the wrong option is focused when a list is entered), *implicit focus* (the focus has to be reset due to changing system conditions) or *explicit focus* (the user resets the focus by turning or pushing the CCE).

For widgets, often additional behavior is specified. One example might be alphabetic scrolling to allow the user to jump to a subgroup of list entries starting with one specific letter. Reports describing that such behavior is either *missing* (specified behavior is not implemented), *wrong* (instead of specified behavior, behavior not specified is implemented) or *extra* (behavior not specified is implemented), are considered as deficient *widget behavior*.

2) Screen Structure Failures: In this failure class, reports are clustered describing the logic to determine the widget objects the screens contain and what data they hold. In automotive infotainment systems, the availability of options depends

Table II
THE DISTRIBUTION OF FAILURES

1. level	2. level	3. level	4. level	distr.
TBC	-	-	-	*7.6 %*
Behavior (Σ: 61.5%)	Screen Transition (Σ: 17.9%)	missing	-	*5.8 %*
		extra	-	*2.9 %*
		wrong	-	*9.2 %*
	Pop-up Behavior (Σ: 11.7%)	missing	-	*3.6 %*
		extra	-	*3.2 %*
		priority	-	*0.5 %*
		wrong	-	*4.4 %*
	Screen Structure (Σ: 13.8%)	screen composition (Σ: 5.4%)	missing	*2.4 %*
			extra	*0.9 %*
			wrong	*2.1 %*
		options offer (Σ: 5.4%)	missing	*2.2 %*
			extra	*1.3 %*
			wrong	*1.0 %*
			order	*0.9 %*
		option gray-out (Σ: 3.0%)	missing	*1.6 %*
			extra	*1.0 %*
			wrong	*0.4 %*
	Widget (Σ: 18.1%)	Behavior (Σ: 14.7%)	missing	*5.1 %*
			extra	*0.9 %*
			wrong	*8.7 %*
		focus (Σ: 3.4%)	initial	*0.9 %*
			implicit	*1.5 %*
			explicit	*1.0 %*
Contents (Σ: 25.1%)	Text (Σ: 15.1%)	design time (Σ: 5.9%)	missing	*1.2 %*
			incomplete	*0.3 %*
			extra	*0.5 %*
			wrong	*3.9 %*
		run time (Σ: 9.2%)	missing	*2.2 %*
			incomplete	*1.1 %*
			extra	*1.0 %*
			wrong	*4.9 %*
	Animation (Σ: 1.8%)	design time (Σ: 0.8%)	missing	*0.4 %*
			extra	*0.1 %*
			wrong	*0.2 %*
			others	*0.1 %*
		run time (Σ: 1.0%)	missing	*0.4 %*
			extra	*0.1 %*
			wrong	*0.3 %*
			others	*0.1 %*
	Symbols & Icons (Σ: 8.2%)	design time (Σ: 2.9%)	missing	*1.5 %*
			extra	*0.2 %*
			wrong	*1.2 %*
		run time (Σ: 5.3%)	missing	*2.2 %*
			extra	*1.0 %*
			wrong	*2.1 %*
Design (Σ: 5.8%)	color	-	-	*1.0 %*
	font	-	-	*0.4 %*
	dimension	-	-	*0.7 %*
	shape	-	-	*0.4 %*
	position	-	-	*2.7 %*
	other	-	-	*0.6 %*

 ISBN: 978-1-61208-233-2

on numerous conditions, such as available devices (e.g., radio tuner available, connected mobile phones, etc.), the current environmental conditions (e.g., car is moving faster than 6 km/h) or even previous interactions (e.g., activating route guidance). These conditions affect, whether options are displayed but cannot be selected (gray-out mechanism) or whether options are listed at all. Failure reports describing that the options are displayed incorrectly are considered as deficient *option offer* or *option gray-out*. The subclass *screen composition* clusters failures related to deficient setup of widgets on screen. Subclasses of this category are *wrong widget* (the wrong widget is displayed), *extra widget* (an unspecified widget is displayed) or *missing widget* (widgets that are specified are absent). *Screen structure* failures are distinguished from the *widget behavior* category as follows: the former represents erroneous selection of widgets such as horizontal or vertical lists, whereas the latter clusters failures of widget behavior itself, such as the scrolling logic or widget state change.

3) Screen Transition Failures: As described above, screens represent one special usage context. The failure class *screen transition* clusters failures occurring when those usage contexts change. One indication for a screen transition is that the widget composition and the displayed options are replaced. With Figure 1 and Figure 2, a screen transition is demonstrated: first, the Radio screen is shown; with activating the option “Tel”, the context changes to the telephone screen of the infotainment system. Subclasses of this category are *missing transitions* (a specified transition does not take place), *extra transitions* (a transition that is not specified takes place) or *wrong transitions* (instead of screen A, screen B is displayed).

4) Pop-up Behavior Failures: With automotive infotainment systems, messages are often overlaid over the regular screen (Pop-up mechanism). Those messages inform users about relevant events or change of conditions. For example, those messages might state that the car has reached the destination of an active route guidance or that hardware has heated up critically. Subcategories are *missing* (the pop-up is not displayed although the respective conditions are active), *extra* (pop-up appears although the respective conditions are not active) and *wrong* (instead of pop-up A, pop-up B is displayed). Additionally, with the pop-up mechanism the priority system is important: a pop-up with higher priority always has to be displayed on top of pop-ups with lower priority. Those failures are clustered in the subclass *priority*.

B. Contents Failures

The next top level category is related to contents. The separation criterion is the type of the content: *symbols & icons*, *animations*, or *text*. In Figure 1, a contents failure would be, if the button for the “Audio” application would have been labeled incorrectly with “Adio” or the globe symbol in the upper right corner of the screen would be a placeholder. In this classification, we distinguish content that is known at *design time* (e.g., the labels of available applications) and content that cannot be defined until *run time* (e.g., displaying the names of available Bluetooth devices). For each of those content types, subclasses for *wrong*, *missing* and *extra content* have been defined.

This category might be confused with the screen structure failure class in the behavior sub tree. For example, a failure report describing that the second button in the main menu is “Blind Text” instead of “Audio” could be categorized as *contents* or *option provision* failure. If pressing the button still leads to a screen transition to the Audio context, the report is considered as deficient contents. If another context is displayed, for example the telephone screen, it would be a deficient option provision.

C. Design Failures

The last top level category clusters reports, which describe design failures. This includes *color* (e.g., focus color is red instead of orange), *font* (e.g., text font is Times New Roman instead of Arial), *dimension* (e.g., a button is higher or broader than specified), *shape* (e.g., a button should be displayed with rounded instead of sharp edges) and *position* (e.g., a label of a button is centered instead of left-aligned). As design failures often were described vaguely, a subcategory for *other* design failures was defined. Ambiguous descriptions were, for example, that wrong arrows, wrong Cyrillic letters or a wrong clock were observed. As it became obvious early, that a low percentage of reports were categorized as design failures, no additional work has been done to clarify this category.

V. Discussion

The requirements defined in Section 3 were met for most failure classes. We intended to cover at least 90% of all defect reports analyzed. Only 7.6% of the reported failures had to be classified as “to be categorized”. Furthermore, we intended to limit the percentage of the classes on the lowest level of the hierarchy to 10%. This could be achieved as well: with 9.2%, the largest category was *behavior - screen transition - wrong*. We intended to allow only 2-5 categories on each hierarchy level. This could not be realized for the design category (6 subclasses). However, due to a very small number of failures classified as design related (5.8%), we did not consider it necessary to restructure this category.

Further, we answered the question, what types of failures are frequent in current GUI software. The results show that the majority (61.5%) are failures related to behavior. This points out the complicated macro and micro behavior in modern infotainment systems. Most of the failure reports are related to missing or wrong individual widget behavior (13.8%), as well as missing or wrong screen transitions (15.0%). The content category is the second biggest top level failure class (25.1%), with erroneous text being the biggest

 ISBN: 978-1-61208-233-2

subcategory (15.1%). The majority (9.2%) is not known until run time. Explanations are (a) that in most infotainment systems information is mainly displayed textually and (b) that testing texts is easier for human testers than comparing symbols or animations in detail. Very few failures (5.8%) describe erroneous design. One explanation might be, that design is hard to test manually. For example, it is a problem to differentiate shades of colors by eye. In addition, most design errors are less critical and might even not be recognized by users. Therefore, testing design might not be of high priority to test planners.

VI. Conclusion and Future Work

In this paper, we answered the question what types of failures can be found in GUI based infotainment systems in the automotive domain today. A failure classification has been developed and applied to more than 3,000 failure reports created during the development of modern automotive infotainment systems at AUDI, Bosch and Mercedes-Benz. 62% of the reports describe failures related to high and low level behavior, 25% of the reports describe failures related to contents and 6% of the reports describe failures related to design. We support not only testers, but the entire GUI development process by pointing out pitfalls leading to gaps between the specification and the implementation. The classification indicates, what aspects need special attention in specification documents and might need to be described more explicitly than is usual today. For roles responsible for the implementation of GUI concepts, this work points out aspects that might be ambiguous and need clarification.

In future research, the suggested classification might be scaled by reducing the maximum percentages of lowest level categories. Thus, some categories have to be differentiated further and additional failure classes have to be defined. Moreover, additional parameters such as “failure criticality”, “predicted number of affected users” or “costs for testing” could be added to the classification. Those aspects are not in focus at the current stage and might influence the choice of test strategies significantly. One could then focus or prioritize testing on those types of failures, which are most critical, based on the frequency and these additional parameters. For this, coverage criteria and prioritization techniques are currently examined, to check, which of them, if any, may be used for our purposes. This classification could be applied to future automotive infotainment systems to analyze change of the failure focus.

Acknowledgment

The authors would like to thank Krishna Murthy Murlidhar, Sven Neuendorf and Jasmin Zieger for their contributions. The research described in this paper was conducted within the project automotiveHMI. The project automotiveHMI is funded by the German Federal Ministry of Economics and Technology under grant number 01MS11007.

References

[1] B. Robinson and P. Brooks, “An initial study of customer-reported gui defects,” in *Software Testing, Verification and Validation Workshops, 2009. ICSTW’09. International Conference on*. IEEE, 2009, pp. 267–274.

[2] C. Bock, “Model-driven hmi development: Can meta-case tools do the job?” in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. IEEE, 2007, pp. 287b–287b.

[3] R. Chillarege, “Orthogonal defect classification,” *Handbook of Software Reliability Engineering*, pp. 359–399, 1999.

[4] N. Li, Z. Li, and X. Sun, “Classification of software defect detected by black-box testing: An empirical study,” in *Software Engineering (WCSE), 2010 Second World Congress on*, vol. 2. IEEE, 2010, pp. 234–240.

[5] J. Børretzen and R. Conradi, “Results and experiences from an empirical study of fault reports in industrial projects,” *Product-Focused Software Process Improvement*, pp. 389–394, 2006.

[6] R. Grady, *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc., 1992.

[7] B. Beizer, “Software system testing techniques,” *New York: Van Norstrand Reinhold*, 1990.

[8] P. Brooks, B. Robinson, and A. Memon, “An initial characterization of industrial graphical user interface systems,” in *Software Testing Verification and Validation, 2009. ICST’09. International Conference on*. IEEE, 2009, pp. 11–20.

[9] M. Kabir, “A fault classification model of modern automotive infotainment system,” in *Applied Electronics, 2009. AE 2009*. IEEE, 2009, pp. 145–148.

[10] J. Ploski, M. Rohr, P. Schwenkenberg, and W. Hasselbring, “Research issues in software fault categorization,” *SIGSOFT Software Engineering Notes*, vol. 32, no. 6, pp. 1–8, November 2007.

[11] “Standard classification for software anomalies,” *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. C1 –15, 7 2010.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA: Addison Wesley, 1995.

[13] S. Stoecklin and C. Allen, “Creating a reusable gui component,” *Softw. Pract. Exper.*, vol. 32, no. 5, pp. 403–416, Apr. 2002.

[14] J. Chen and S. Subramaniam, “Specification-based testing for gui-based applications,” *Software Quality Journal*, vol. 10, pp. 205–224, 2002.

 ISBN: 978-1-61208-233-2

A Holistic Model-driven Approach to Generate U2TP Test Specifications Using BPMN and UML

Qurat-ul-ann Farooq*, Matthias Riebisch†
*Department of Software Systems / Process Informatics, Ilmenau University of Technology
98684 Ilmenau, Germany
{qurat-ul-ann.farooq}@tu-ilmenau.de
†Department of Informatics, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
riebisch@informatik.uni-hamburg.de

***Abstract*—Testing process-based information systems is cost intensive and challenging due to rapid technological advancement and increasing complexity of processes. A number of existing process-based test generation approaches use process code for test generation. They operate on lower levels of abstraction and start the test activity later in the development cycle, which is not feasible. Other model-based testing approaches focus only on the individual behavior of a process. They do not consider the structural aspects and process interactions; thus, are not able to capture different test views. In this paper, we present a model-driven test generation approach that uses UML class and component diagrams to model the structural aspects, and BPMN collaboration diagrams to model the collaborative behavior of business processes. Models from both views are used as input to generate the test specifications, which are expressed as of UML 2 Testing profile (U2TP) elements. To identify the correspondences between the process structure, behavior and the test view, we analyze the semantics of UML, BPMN, and U2TP. We developed mapping rules to realize these correspondences for the generation of U2TP test specifications from UML and BPMN models. Our mapping rules are implemented as model transformations using the VIATRA model transformation framework. We illustrate the approach using an example scenario to demonstrate its applicability.**

***Keywords-MDA*; *Model-driven Testing*; *BPMN*; *U2TP*; *Business Process Test Generation*.**

I. INTRODUCTION

Testing enterprise software systems is essential to ensure the quality of the systems supporting the underlying business processes. However, due to the increasing complexity of processes and rapid technological advancement, testing requires high effort and huge investments. Furthermore, early testing is required to save project costs. This can be achieved by deriving the test specifications from the process design specifications. However, most of the existing business process-based testing approaches use low level artifacts for test generation, such as process code or web service description language (WSDL) [1][2][3][4], which is often not available in the early phases of software development.

To deal with this issue, Model-driven testing (MDT) [5] for enterprise business processes has been introduced [6][7]. It enables the test generation from the high level process models instead of process code; thus, enabling testing activity in the early phases of the development life cycle. This results in reduced costs and cross platform portability of the test suites. Model-driven testing uses the concept of model transformations to transform the platform independent design models into platform independent test suites. Later, concrete test specifications or test code can be generated from these test models [5]. Hence, to support the model-driven test generation for process-based information systems, there are three major requirements; (1) the availability of a platform independent process modeling language, (2) the availability of a test modeling language to support test visualization and documentation, and (3) the support for model transformations for the test generation.

In this paper, we present a model-driven test generation approach for process-based information systems. To meet the first requirement, the artifacts required to model different views of process-based information systems are to be analyzed [8]. These different system views include the *Process View*, *Resource/Structure View*, *Behavior View*, and the *User Interface view* [8][9]. The existing model-based testing approaches in the literature only focus on the behavioral view of the processes for the test generation. However, the information from other system views can also be used to generate parts of the test specification. For example, the structural system view can be modeled using the UML class diagram and component diagram. The information about the system structure can be obtained from these models to generate the test architecture and test data [10].

In our approach, to model the process and behavior view we use BPMN collaboration diagrams, while modeling the structural view of the system using the UML component and class diagrams. Both UML and BPMN are standards from the *Object Management Group* (OMG)[11], [12]. We use the process modeling guidelines from the *UML-based Web Engineering* (UWE) [13] approach and the *Service Oriented Architecture Modeling Language* (SoaML) [14].

 ISBN: 978-1-61208-233-2

To fulfill the second requirement, which is support for the test modeling, we use U2TP [10], which is also a test modeling standard from OMG. U2TP covers several important test modeling issues, such as, modeling the *Test Architecture*, *Test Behavior*, *Test Data*, and *Test Time*. In this paper, we focus primarily on the generation of U2TP test architecture and test behavior for business process-based testing. Finally, to support the third requirement, we identify the correspondences between the design artifacts and test models and develop the mapping rules using these corresponding elements. These mappings are defined by analyzing the semantics of the BPMN collaboration diagrams, UML class and component diagrams, and the U2TP test models. The mapping rules are implemented in the form of model transformations. We used *Viatra Transformation Control Language* (VTCL), which is a model transformation language provided by the VIATRA model transformation framework [15] for the implementation of our transformations. The rest of this paper is organized as follows.

Section II discusses the related work and analyzes the strengths and weaknesses of business process-based testing approaches. Section III provides an overview of our approach. Section IV discusses our model-driven test case generation approach in detail and also presents the mapping rules for the test generation. Section V conferes the implementation details and Section VI illustrates the application of the approach on an example scenario. Finally, Section VIII concludes the paper and discusses the future directions of our work.

II. RELATED WORK

Most of the process-based test generation approaches derive the test cases from the process code. These approaches either generate test paths directly from the code, based on data and control flow information [16], [17], or translate the code into formal specifications languages like Petri-nets [2], [3], [18], to perform the model checking and test derivation. One of the major disadvantages of these approaches is that the tests cannot expose the deviations from the functional specifications, as the tests are directly derived from the code. Moreover, the testing activity can only be started after the development is complete, which increases the cost as well as the time allocated to the testing phase.

Werner et al. [4] use the WSDL process specifications for the test generation. They only consider the interfaces of the processes; thus, generate only black box test cases from them and do not consider the internal control flows or data flows of the system.

There are a few approaches focusing on model-based test generation for process-based systems. Bakota et al. [1] use a graph like notation for the process specification, where the nodes of the graph represent activities with distinct input and output parameters. The category partition method is used to derive test data for individual activities. They generate the test paths based on the data values and then convert them into the test frames. The approach presents interesting concepts but targets only the data-based process specification languages. The process models in BPMN support many additional activity types and events and they should also be considered during the test generation.

Heinecke et al. [19] present an approach for test generation, where a process is specified using activity diagrams. However, like Bakota et al, Heinecke et al. also do not support event-based process specifications for the test generation. The major distinctions of our approach from the approaches of Heinecke et al. and Bakota et al. are that we not only support events and various activity types during the test generation, we also use the concept of holistic modeling and test generation. Thus, we focus not only on the behavioral aspects, but also consider the structural aspects of the tests during the test generation.

Yuan et al. [7] present a model-driven test generation approach for process-based systems. Our approach is also using the same foundations as Yuan et al., however, their work is only an initial idea and lacks details regarding the test generation activities and the rules. Moreover, their work focuses only on the test architecture generation and lacks the test behavior generation aspect.

The model-based test generation approaches for testing business processes discussed in this section, do not consider a holistic view of the system during the test generation and rely only on behavioral artifacts, such as graphs or activity diagrams. However, as discussed earlier, the artifacts representing different views of a process-based system can provide input to generate different test views and thus, should be considered during the test generation. In the next section, we present our holistic model-driven approach that uses the artifacts from the structural, behavioral and process views of the system for the test generation.

III. OVERVIEW OF THE APPROACH

To perform effective model-based testing of process-based systems, the first step is to select an appropriate modeling methodology, and model the system architecture and business processes. After that, a quality assurance (QA) analyst can review the models for testability. This includes checking the models for completeness and validating any constraints required to generate the test specification. In the next step, a test generation tool can generate abstract test specifications using these models. We discuss these steps in detail in the following subsections.

A. Business Modeling using UWE and SoaML

As discussed earlier, we model the system architecture and the processes using the UWE [13] and SoaML [14] approaches. Since the focus of this paper is on the test generation by analyzing process interactions and structural

 ISBN: 978-1-61208-233-2

aspects of the system, we briefly discuss the artifacts we use from these approaches and their specifics.

The UWE approach originally uses an activity like model to model the processes. However, instead of using that, we use BPMN collaboration diagrams and process diagrams to model the behavior of interacting and atomic processes. To model the structural aspects of the system, we use the UML class diagram. Using the UWE approach, each process itself is modeled as a class with a stereotype «ProcessClass» and the data and resources of the system are modeled as classes with a stereotype «entity». From the SoaML approach, we use the component models to define the high level structure of process-based systems.

As discussed earlier, once the models are complete, they should be analyzed for testability. Testability is an important property of the model because a less testable model can results in poor test cases. The testability requirements of our approach are discussed in the following.

- → Completeness: The artifacts used as input are complete and all the processes have their corresponding structures defined in the corresponding UML class diagram.
- → Since handling of multiple entry and exit points is complex, we restrict a collaboration diagram to exactly one start and end node within one pool to reduce additional complexity
- → To ease the testing, we restrict each Pool in the collaboration diagram, to have a corresponding component in the component diagram with well defined interfaces for access. A lane can map to a process class or a service class in the UML class diagram.

Once the models are complete and reviewed by a QA expert for testability, they can be used to generate the test specifications. In the next section, we discuss the foundations of our model-driven test generation approach, and in the later sections we elaborate the approach in more detail.

B. The Abstract Test Specification Generation

As discussed earlier, we use U2TP for the specification of the test architecture and test behavior. The following tasks are to be performed for model driven test generation using our approach.

1) Generate the test architecture by analyzing the structural system models, which are in our case UML class and component diagrams.
2) Transform the test architecture into a class diagram to support the test visualization.
3) Generate the test behavior from BPMN collaboration models in the form of test paths. These test paths can be constructed using path analysis algorithms from the graph-based test generation approaches using a certain coverage criterion.
4) In the next step, the generated paths are transformed into UML activity diagram paths. This transformation satisfies the second requirement stated in the introduction section of this paper. At this stage the tester can analyze each individual test path and add the additional information, such as test verdicts etc.

Test data can also be generated by using process constraints and data resources defined in the structural models; however, the test data generation is out of scope of the current paper. In the next section, we discuss our model-driven test generation process and the above discussed tasks in detail .

IV. THE MODEL-DRIVEN TEST GENERATION PROCESS

To generate the test specifications, we adapted the classic model-driven test generation process by Dai et al. [5]. This

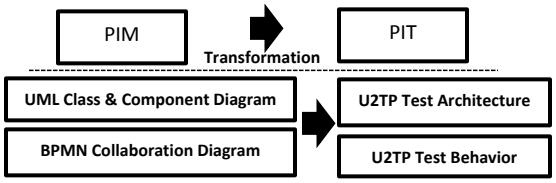


Figure 1. The Example Software Architecture of a Banking Example

process involves the transformation of a platform independent model (PIM) into a platform independent test model (PIT). The upper part of Figure 1 shows this pattern from Dai et al. [5], where a PIM model is transformed to a PIT model.

The lower part of Figure 1 shows our adaptation of the process for the generation of U2TP test architecture and test behavior using UML and BPMN models. The test models in U2TP should cover the structural and behavioral aspects of the test system. These aspects can be derived from the structural and behavioral specification of the system. To do this, we transformed the platform independent UML class and component diagrams representing the structure and resources of the processes into U2TP test architecture models. For the test behavior generation, the platform independent BPMN Collaboration diagrams and process diagrams are transformed into U2TP test behavior, which represents the abstract platform independent test specification.

To define these transformations, the mapping relations between the elements of source and target languages are to be identified. We identify these mapping relations by analyzing the correspondences between the relevant elements of these languages. The elements and semantics of the UML class and component diagrams, BPMN process and collaboration diagrams, and U2TP test models are defined in their respective meta-models[11], [12], [10].

We define the mapping relations in the form of mapping rules. In principle, a mapping rule realizes a mapping relation that represents a correspondence between the relevant source model and the target model of a particular transformation.

 ISBN: 978-1-61208-233-2

We define a mapping rule as a 4 tuple (*Source Element*, *Target Element*, *Rule Preconditions*, *Rule Postconditions*), where the *Source Element* is an element of the source PIM model, i.e., UML or BPMN, and the *Target Element* is an element in the target PIT model, i.e., U2TP. The *Preconditions* define any constraints for the execution of the rule and the *Postconditions* define the changes in the state of the target test models, such as the addition of new elements. An example of such a rule is presented in Listing 1. In the next subsections, we discuss both U2TP test architecture and test behavior generation activities, and the mapping rules we developed to define the transformations in detail.

A. Generation of U2TP Test Architecture

The test architecture is a representation of the structural aspects of a test system. To define the test architecture, U2TP provides several elements. These elements are: *System Under Test* (SUT), *Test Arbiter*, *Test Scheduler*, *Test Context*, and *Test Components*. To specify the test architecture to test a process, these elements are required to be specified. For this purpose, we analyze the elements representing the process structure and derive the test structure from these elements.

Source Element: Class , Test package: CD^{SA}
Target Element: Class , Association : CD^{TA}
Preconditions:
1. $\exists Class.C \in CD^{SA}$
2. $\exists$ Stereotype . ProcessClass $\in$ C
3. $\exists$ U2TP. TestPackage $(TP \mid TP \in CD^{TA})$;
4. C $\in$ TP
5. $\exists$ BPMN. Collaboration Diagram $(CD \mid CD \in C)$;

Postconditions:
6. $\exists Class.T \in CD^{TA}$
7. T . Name=C . Name
8. $\exists$ Stereotype .SUT $\in$ T
9. $\exists Dependency.Import.A \in CD^{TA}$
10. $A \in TP, A \in T$

Listing 1. A Mapping Rule for SUT

To represent the test architecture, U2TP proposes the UML class diagram notation with stereotypes for the U2TP elements. We refer to the class diagram representing the test architecture as CD^{TA}, and the class diagram representing the system architecture as CD^{SA} in the our mapping rules. In the following, we discuss, how we derived the test architecture elements from the UML class and component diagrams, which represent the system structural aspects.

In the UWE process modeling approach, a class corresponding to a collaborative process is represented by a stereotype «ProcessClass». Since a *Process Class* defines the process representing interactions between several participants, it is a candidate class for testing the collaborative process. This means that this class can be treated as a system under test or process under test. Hence, we map each *Class* with a stereotype «ProcessClass» in the CD^{SA} to a *System under Test (SUT)* in the CD^{TA}. This mapping is realized by the mapping rule presented in Listing 1.

According to the rule, there are three preconditions in the *Preconditions* part. The first precondition (Line 1 and 2) states that a class named C with the stereotype «ProcessClass» should exist in the system architecture class diagram CD^{TA}. The second precondition (Line 3 and 4) indicates that a corresponding *Test Package* should be present in the test architecture class diagram, and the third precondition (Line 5) ensures that a BPMN collaboration diagram is present that corresponds to the process class C. The presence of a test package is required due to the reason that each test related element of a particular process class is packaged into one particular test package for better test organization [5].

In the postcondition part, a type *Class* T with a stereotype «SUT» corresponding to the class C is created in the test architecture class diagram CD^{TA} (Line 6, 7, and 8). Since elements of a test package require the SUT for the test execution [5], an import dependency A is created between the test package TP and the SUT class T (Line 9 and 10).

To map the other elements in the test architecture, such as the *Test Context* and the *Test Components*, we developed mapping rules like the one shown Listing 1. The rules to generate the test components are more complex as they require additional information from the UML component diagram and the BPMN collaboration diagram. Due to the space limitations, it is not possible to include all of these rules in the paper; however, a subset of these rules is available on our website for additional reading[20]. After establishing the mappings for the U2TP test architecture, the next step is to develop the mappings to generate U2TP test behavior, which is explained in the next section.

B. Generation of U2TP Test Behavior from BPMN Collaboration Diagrams

In this section, we discuss our test behavior generation process and the mapping rules we developed for the test behavior generation. Our test behavior generation process comprises of two major tasks; the generation of test paths from BPMN collaboration diagrams, and the transformation of these test paths into UML activity diagram test cases. The generation of the test paths is dependent on the selected coverage criteria. However, the activity of mapping the test paths onto the activity diagram test cases should be independent of the coverage criteria and test path generation strategies. So that it can be reused with different coverage criteria and path generation activities. To enable such reuse, we split our test generation process into three sub activities, as depicted in Figure 2.

According to Figure 2, the first activity prepares the BPMN collaboration diagram for test path generation by extending it with some information required by the path extraction algorithms. The algorithm we use in this work uses the distance information for the selection of test paths. Thus, the distance of each node from the end node is computed, and the nodes are annotated with this information.

 ISBN: 978-1-61208-233-2

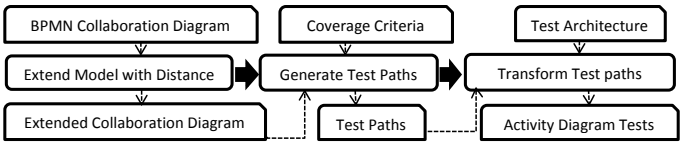


Figure 2. Test Behavior Generation Activities

The output of this activity is a diagram extended with distance information. In the second activity, the test paths are extracted from the extended collaboration diagram based on some coverage criterion. In this paper, we use the branch coverage criterion, which is further explained in the next subsection. Finally, during the third activity, the generated paths are transformed into UML activity diagrams to support the visualization and test documentation.

In U2TP, the test behavior can be specified using either the UML sequence diagrams or the UML activity diagrams. In this work, we selected the UML activity diagram notation for the test behavior specification due to its natural similarity to the process specifications. As discussed earlier, the mapping activity to map the test paths onto UML activity diagrams is independent of the test path generation activity; thus, it can be reused with multiple coverage criteria. In the following subsections, we discuss the test path generation and the details of mapping the test paths onto UML activity diagrams.

1) Generate Test paths: To enable the test path generation from BPMN collaboration diagrams, we use the branch coverage criterion. This criterion covers all the gateways in the diagram for all possible outcomes, and covers each loop once. For computing the test paths, we first compute the shortest path in the collaboration diagram by using the Dijkstra algorithm [21]. The reasons for using the Dijkstra algorithm for computing the shortest path first is that in our test suite, the first test case will always contain the shortest execution path. In the case of limited testing budget, execution of this test case can provide the confidence about at least one process path execution with the minimum cost overhead.

For the calculation of other paths in the diagram we use a Depth First Search (DFS) algorithm with backtracking [22]. The reason for selecting the DFS is its ability to cover all the branches of a graph by visiting all child nodes of a node and to backtrack, when the end node or an already visited node is found. When a node is added to a path, all the information of the node is also copied. If the node corresponds to a send, receive, or service task, the information about the related pools is also copied with that task. BPMN collaboration diagrams can contain parallelism by using the parallel gateways in the diagram. To deal with it, one of the simplest strategies can be to place all the branches of the gateway in a sequential order [1]. However, more complex execution strategies can exist. We are treating all the branches of a parallel gateway as one test case and defer the decision of their execution strategies to the concrete test generation activity. After the generation of the test paths, the next activity is to map each test path onto a UML activity diagram, which is discussed in the next section.

2) Transform Test Paths to UMLAD: The transformation of the test paths extracted in the previous steps to a UML activity diagram requires the identification and definition of mappings between the elements of BPMN collaboration diagram and UML activity diagram. These mappings can be identified by analyzing the constructs of BPMN collaboration diagram and the corresponding elements in the UML activity diagram. Based on these correspondences, we develop the mapping rules, which realize the correspondences or mapping functions.

```
Source Element: Collaboration: Message Start Event
Target Element: Activity: Initial Node, AcceptEventAction,
    SignalEvent, Controlflow
Preconditions:
∃ BPMN.Pool (X)∨ BPMN.Lane(X);
∃BPMN.EventStartMessage(Start | Start ∈ X);
∃UML.ActivityPartition(AP | AP ∈ X);
Postconditions:
∃ UML.InitialNode(I | I ∈ AP);
∃ UML.AcceptEventAction(accept | accept ∈ AP, name(accept)
    =name(Start)+'AcceptAction');
∃ UML.OutputPin(OP | OP ∈ accept);
∃ UML.Trigger(trigger| trigger ∈ accept, name(trigger)=
    name(Start)+'Trigger');
∃ UML.SignalEvent(te | te ∈ trigger);
∃ ControlFlow(cf | cf(I, accept));
```

Listing 2. An excerpt of the Mapping Rule for Message Start Event

BPMN collaboration diagrams comprise of a set of start, end, and intermediate nodes, tasks and activities, pools and lanes, control flows and message flows, gateways, and several data elements. A mapping function or mapping rule is required to be defined between each of these elements and their corresponding activity diagram elements.

In the following, we provide an example of a mapping rule that maps a start event in BPMN collaboration diagram to the corresponding activity diagram elements. The start and end nodes in a BPMN collaboration diagram can be mapped to the *Initial Node* and *Final Node* in the UML activity diagram. However, BPMN collaboration diagrams have many different types of start and end nodes, such as, *Message Start Event*, *Empty Start Event*, *Timer Start Event*, and many others. Since the UML activity diagram has only one Initial Node type, it can be mapped to only one type of start event in BPMN collaboration diagrams, i.e., *Empty Start Event*. For the remaining events, more complex patterns are required to be identified in the UML activity diagrams.

An example of such event is the *Message Start Event*.

 ISBN: 978-1-61208-233-2

The mapping rule for the *Message Start Event* is depicted in Listing 2. According to Listing 2, a *Message Start Event* can be mapped to an *Initial Node* followed by an *Accept event Action* in a UML activity diagram. The preconditions stated in the preconditions part specify that in the BPMN collaboration diagram, either a *Pool* or a *Lane X* exists and the *Message Start Event* belongs to that *Pool/Lane*. The next precondition is that an *Activity partition* exists corresponding to the *Pool/Lane X*. We map a *Pool* or a *Lane* in the BPMN collaboration diagram to an *ActivityPartition* element in the UML activity diagram. This *ActivityPartition* is a container for other elements such as *Tasks*, *Events* and *ControlFlows* in the activity diagram.

The postcondition part is rather complex. It states that an *Initial Node* in the UML activity diagram is created, which is followed by an *AcceptEventAction*. The trigger of this action is a *Signal Event*. A *Control Flow* is added between the *Initial Node* and *Accept Event Action* to maintain the sequential dependency between both. Due to the space limitations, it is not possible to discuss all the elements and their respective mapping rules in this paper. However, Table I depicts a subset of the mappings between the elements of BPMN collaboration diagram and UML activity diagram. A complete set of mapping rules is available at our project website [20]. After each test path is mapped to an activity

Table I
MAPPING SUBSET:COLLABORATION ONTO ACTIVITY DIAGRAM

Collaboration Elements	Activity Elements
Pool	ActivityPartition
SequenceFlow	ControlFlow
Gateway	DecisionNode
EmptyTask/None	Action
SendTask when target is a non-service Task or Pool	SendSignalAction
SendTask when target is a service Task	CallOperationAction
ReceiveTask	AcceptEventAction
A Task calling a ServiceTask	CallOperationAction
EmptyStartEvent/EmptyEndEvent	InitialNode/FinalNode

diagram, the test case definitions are added to the *Test Context* class as test operations in the test architecture class diagram. The interface of these operations can be generated by analyzing input data required by each path.

V. TRANSFORMATION IMPLEMENTATION USING VIATRA

We implemented our test generation approach and the mappings using the model transformation framework VIATRA [15]. VIATRA provides a rule-based language *VIATRA Textual Command Language* (VTCL) for the implementation of model transformation rules. The rule-based structure of VIATRA makes it suitable for the implementation of our mapping rules.

The basic construct of VTCL is a *Graph Transformation Rule* (GT-Rule). A *GT-Rule* is comprised of a precondition part, a postcondition part and an action part. When a precondition pattern of a *GT-Rule* is matched in the source model, it creates an image of the postcondition pattern in the target model. Our mapping rules can be seen as an abstract form of a *GT-Rule* and are easily translatable to the concrete executable *GT-Rules*. The models we use to implement the transformations conform to the meta-models of UML, BPMN, and U2TP.All the meta models are implemented as Eclipse plugins using the EMF framework [23].

```
gtrule singlePool(inout Pool, inout Activity)={
  precondition pattern isPool(Pool)={
   bpmn.metamodel.bpmn.Pool(Pool);
   neg find isLaneInPool(Pool,Lane);}
  postcondition pattern matchPartitionToPool(VisKind,
       Partition,Pool,Activity,String)={
   bpmn.metamodel.bpmn.Pool(Pool);
   uml.metamodel.uml.ActivityPartition(Partition) in
        Activity;
   uml.metamodel.uml.VisibilityKind(VisKind) in Partition;
   uml.metamodel.uml.NamedElement.name(NamedElem,Partition
        ,String);
   uml.metamodel.uml.NamedElement.visibility(Vis,Partition
        ,VisKind);}
  action{
   rename(Partition,name(Pool));
   setValue(VisKind,"public");}}
```

Listing 3. An excerpt of the *GT-Rule* in VTCL

Listing 3 presents a *GT-Rule* for the mapping *"Pool maps to ActivityPartition"*, as depicted in the first row of Table I. The graph transformation rule, *"singlePool"* shown in Listing 3 takes a *Pool* and an *Activity* as input. The input *Activity* is the base element of the activity diagram meta-model. The rule *singlePool* consists of a precondition pattern and a postcondition pattern. The precondition pattern states that *Pool* is an element in the BPMN meta-model. The second line of the precondition pattern checks if there is a lane inside the pool or not. The postcondition pattern creates an *ActivityPartition* inside the element *Activity*, and instantiates its properties. In the action part of the *GT-Rule*, name of the *Pool* is assigned to the created *ActivityPartition*, and the property "visibilitykind" is specified as public. A prototype Eclipse plug-in implementing the transformations is available at our project website[20].

VI. CONCEPT ILLUSTRATION BY EXAMPLE

To illustrate the applicability of our approach, we introduce an example credit request scenario and apply our approach on it. The left side of Figure 3 depicts an excerpt of the class diagram of the credit request application. The diagram contains a class *HandleCreditRequestProcess* with a stereotype «ProcessClass». A part of the collaboration diagram corresponding to this process class is depicted on the right hand side of Figure 3. Figure 4 represents the U2TP test architecture class diagram. The class *HandleCreditRequestProcess* with the stereotype «SUT» represents the system under test, and is derived by applying the mapping rule in Listing 1. The SUT class has an import dependency to the test package *financial.credit.handlecreditrequestprocess.test*

 ISBN: 978-1-61208-233-2

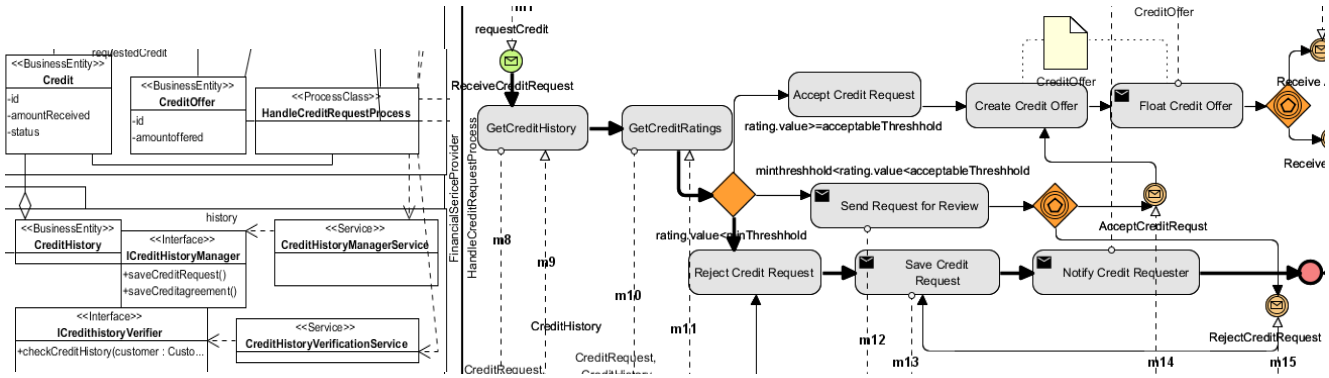


Figure 3. An Excerpt of the Credit Request Collaboration and Component Architecture

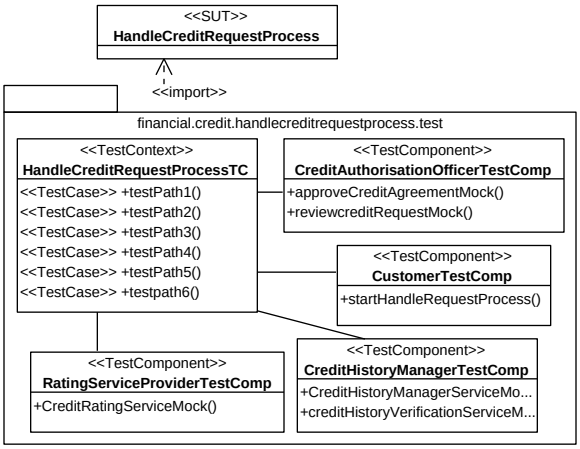


Figure 4. The Credit Request Test Architecture

as defined in the postcondition of the mapping rule in Listing 1.

The classes with the stereotype *TestComponent* shown in Figure 4 are the test components, which are required to test the process *HandleCreditRequestProcess*. These test components are derived from the *Pools* and *Lanes* of the collaboration diagram and their corresponding structural specifications defined in the class and component diagrams.Due to the limited space, these pools are not shown in the collaboration illustrated by Figure 3.

An example test component in Figure 4 is the *CreditHistoryManagerTestComp*, which is responsible of processing the message calls sent by the tasks *GetCreditHistory* and *SaveCreditRequest* in the *HandleCreditRequestProcess*. The test components are derived from the pools receiving the messages.The test component *CreditHistoryManagerTestComp* mocks the two services *CredithistorymanagerService* and *CredithistoryVerificationService*. These services are shown as service classes in the class diagram shown in figure 3. The test package also contains the *Test Context* class and the required test components. The test context class contains six test cases, which are generated by applying the test generation steps explained in the Section IV-B. One of such test paths is depicted with the bold control flow in Figure 3. The UML activity diagram test case corresponding to this test path is depicted in Figure 5.

In the activity diagram, the service tasks are assigned to *CallOperationAction*, send tasks are assigned to the *SendSignalAction*, and receive tasks are assigned to the *AcceptEventAction*.These mappings are consistent with the mappings shown in Table I. The sent and received messages are assigned to the ports of the respective actions in the UML activity diagram test case. For the sake of simplicity, no data flows were shown in figure 3, and no *ActivityPartitions* are shown in the test case shown in Figure 5.

VII. Acknowledgment

We thank Stefan Gross for developing the prototype discussed n section V as part of his masters thesis.

VIII. Conclusion and Future Work

In this paper, we presented a holistic model-driven test generation approach for testing business process-based information systems. For the test generation, first we transform the elements of UML class and component diagrams into U2TP test architecture elements. After that, we generate the test behavior by transforming the BPMN collaboration diagrams into test paths and then transforming these test paths into U2TP activity diagram test cases. To do this, we analyzed the elements of BPMN collaboration diagrams, UML class and component diagrams, and U2TP test models to identify the corresponding elements and then developed mapping rules based on them. We implemented the mapping rules in a prototype, using model transformations provided by the VIATRA framework. One of the benefits of our approach is that we used models for the test generation as well as the test specification, which results in support for early testing and better test documentation. A limitation of our work is that at present we do not support the test data by analyzing the data elements in the BPMN models and their corresponding structures. However, this is part of our ongoing work and we plan to address this issue in the future.

 ISBN: 978-1-61208-233-2

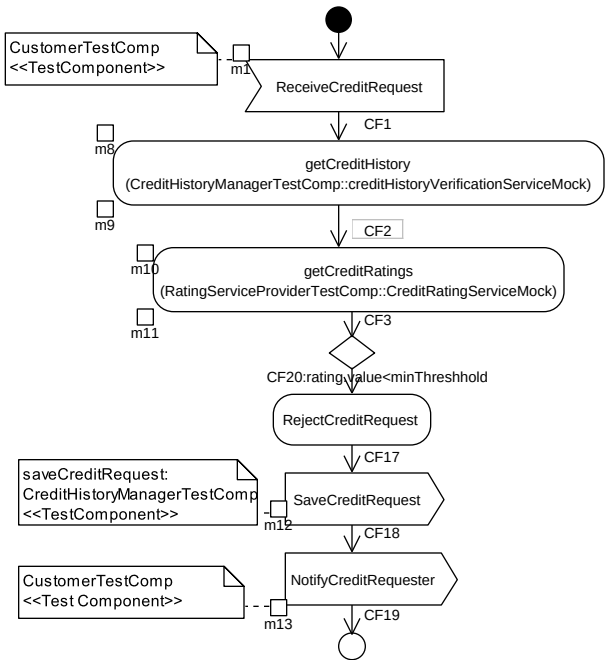


Figure 5. A Test path of the Credit Request Process as UML Activity Diagram Test Case

REFERENCES

[1] T. Bakota, A. Beszédes, T. Gergely, M. I. Gyalai, T. Gyimóthy, and D. Füleki, "Semi-automatic test case generation from business process models," 11th Symposium on Programming Languages and Software Tools, 2009.

[2] J. Garcia-Fanjul, J. Tuya, and C. de la Riva, "Generating test cases specifications for BPEL compositions of web services using SPIN," in *Proceedings of the International Workshop on Web Services: Modeling and Testing*, 2006, pp. 83–94.

[3] Y. Zheng, J. Zhou, and P. Krause, "A model checking based test case generation framework forweb services," in *Proceedings of the International Conference on Information Technology*, 2007, pp. 715–722.

[4] E. Werner, J. Grabowski, S. Troschutz, and B. Zeiss, "A TTCN-3-based web service test framework," in *Workshop on Testing of Software - From Research to Practice*, 2008.

[5] Z. Dai, "An approach to model-driven testing: Functional and real-time testing with UML 2.0, U2TP and TTCN-3," Ph.D. dissertation, TU Berlin, 2006.

[6] A. Stefanescu, S. Wieczorek, and A. Kirshin, "MBT4Chor: a Model-Based testing approach for service choreographies," in *Model Driven Architecture - Foundations and Applications*, 2009, pp. 313–324.

[7] Q. Yuan, "A model driven approach toward business process test case generation," *10th International Symposium on Web Site Evolution*, pp. 41–44, 2008.

[8] M. Penker and H. Eriksson, *Business Modeling With UML: Business Patterns at Work*, 1st ed. Wiley, Jan. 2000.

[9] D. Auer, V. Geist, W. Erhart, and C. Gunz, "An integrated framework for modeling Process-Oriented enterprise applications and its application to a logistics server system," in *2nd International conference on Logistics and Industrial Informatics*, Sep. 2009, pp. 1 –6.

[10] OMG, "UML2 Testing Profile," OMG Docment Formal/ptc/2011-07-20, Object Management Group, July 2005. [Online]. Available: http://www.omg.org/spec/UTP/1.0/

[11] ——, "Business Process Model and Notation (Beta 2)," Object Management Group, June 2010. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/Beta2/PDF/

[12] ——, "UML 2.0 superstructure specification," OMG document formal/2007-02-03, Object Management Group, 2007. [Online]. Available: http://www.omg.org/docs/formal/07-02-03.pdf

[13] N. Koch, A. Kraus, C. Cachero, and S. Meliá, "Integration of business processes in web application models," *J. Web Eng.*, vol. 3, no. 1, pp. 22–49, 2004.

[14] A. Sadovykh, P. Desfray, B. Elvesaeter, A.-J. Berre, and E. Landre, "Enterprise architecture modeling with SoaML using BMM and BPMN - MDA approach in practice," in *6th Central and Eastern European Software Engineering Conference*, 2010, pp. 79 –85.

[15] VIATRA2, "Viatra2, visual automated model transformations framework," Availabe at: http://www.eclipse.org/gmt/VIATRA2/, June 2011.

[16] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang, "BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach," in *17th International Symposium on Software Reliability Engineering*, 2006, pp. 75 –84.

[17] Y. Yuan, Z. Li, and W. Sun, "A graph-search based approach to bpel4ws test generation," in *International Conference on Software Engineering Advances*, 2006, p. 14.

[18] W.-L. Dong, H. Yu, and Y.-B. Zhang, "Testing BPEL-based web service composition using high-level petri nets," in *10th IEEE International Enterprise Distributed Object Computing Conference*, 2006, pp. 441 –444.

[19] A. Heinecke, T. Griebe, V. Gruhn, and H. Flemig, "Business process-based testing of web applications." ser. Lecture Notes in Business Information Processing, vol. 66, 2010, pp. 603–614.

[20] "B2u project website," Available at: http://www.theoinf.tu-ilmenau.de/ qurat/B2UProject/subsite/index.htm, Last Accessed: 09.11.2012, 2012.

[21] J. Sneyers, T. Schrijvers, and B. Demoen, "Dijkstras algorithm with fibonacci heaps: An executable description," in *20th Workshop on Logic Programming*, 2006, pp. 182–191.

[22] B. Awerbuch, "A new distributed depth-first-search algorithm," *Information Processing Letters*, vol. 20, no. 3, pp. 147 – 150, 1985.

[23] EMF, "Eclipse Modeling Framework ," Last Accessed: 09.11.2012. [Online]. Available: http://www.eclipse.org/modeling/emf/?project=emf

 ISBN: 978-1-61208-233-2

Diagnosability Analysis for Self-observed Distributed Discrete Event Systems

Lina YE and Philippe DAGUE
Univ Paris-Sud, LRI, CNRS
Email: lina.ye@lri.fr, philippe.dague@lri.fr

***Abstract*—Diagnosability is a crucial property that determines, at design stage, how accurate any diagnosis algorithm can be on a partially observable system and, thus, has a significant impact on the performance and reliability of complex systems. Most existing approaches assumed that observable events in the system are globally observed. But, sometimes, it is not possible to obtain global information. Thus, a recent work has proposed a new framework to check diagnosability in a system where each component can only observe its own observable events to keep the internal structure private in terms of observations. However, the authors implicitly assume that local paths in components can be exhaustively enumerated, which is not true in a general case where there are embedded cycles. In this paper, we get some new results about diagnosability in such a system, i.e., what we call joint diagnosability in a self-observed distributed system. First, we prove the undecidability of joint diagnosability with unobservable communication events by reducing Post's Correspondence Problem to an observation problem. Then, we propose an algorithm to check a sufficient but not necessary condition of joint diagnosability. Finally, we briefly discuss about the decidable case with observable communication events.**

***Index Terms*—diagnosis; joint diagnosability; finite state machine.**

I. INTRODUCTION

Over the latest decades, with more performance requirements imposed on the complex systems, they are subject to more errors. However, it is unrealistic to detect faults manually for such systems. Automated diagnosis mechanisms are thus required for large distributed applications. Generally speaking, diagnosis reasoning aims at detecting possible faults explaining the observations. The efficiency of diagnosis reasoning depends on system diagnosability, which is a crucial property that determines at design stage how accurate any diagnosis algorithm can be on a partially observable system. The systems we discuss here are Discrete Event Systems (DES).

Some existing works analyzed diagnosability in a centralized way ([1], [2], etc.), i.e., a monolithic model of a given system is hypothesized, which is unrealistic due to combinatorial explosion of state space. This is why very recently distributed approaches began to be investigated ([3], [4], etc.), relying on local objects. However, all these approaches assumed that observable events in the system are globally observed. But sometimes it is not possible to obtain global information. Then, Ye et al. [5] has proposed a new framework to check diagnosability in a system where each component can only observe its own observable events to keep the internal structure private in terms of observations. However, the authors implicitly assume that local paths can be exhaustively enumerated, which is not true in a general case where there are embedded cycles. In this paper, we generalize this work to get some new results about the diagnosability of what we call self-observed distributed systems, where observable events can only be observed by their own component.

We make several contributions in this paper. First, we extend diagnosability of globally observed systems to what we call joint diagnosability of self-observed systems and then to prove its undecidability with unobservable communication events. Secondly, we propose an algorithm for testing a sufficient condition, where we obtain pairs of local trajectories in the faulty component, such that for each pair only one trajectory contains the fault but both have the same observations, and then check their global consistency through two phases. We provide the proof that it is a sufficient condition and point out why it is not necessary. Thirdly, the decidable case where communication events are observable is discussed.

II. PRELIMINARIES

In this section, we model self-observed distributed DES and then recall joint diagnosability features [5].

We consider a self-observed distributed DES composed of a set of components $\{G_1, G_2,..., G_n\}$ that communicate by communication events, where each component can only observe its own observable events. Such a system is modeled by a set of finite state machines (FSM), each one representing the local model of one component. The local model of a component G_i is a FSM, denoted by $G_i = (Q_i, \Sigma_i, \delta_i, q_i^0)$, where Q_i is the set of states; Σ_i is the set of events; $\delta_i \subseteq Q_i \times \Sigma_i \times Q_i$ is the set of transitions; and q_i^0 is the initial state. The set of events Σ_i is partitioned into four subsets: Σ_{i_o}, the set of locally observable events that can be observed only by their own component G_i; Σ_{i_u}, the set of unobservable normal events; Σ_{i_f}, the set of unobservable fault events; and Σ_{i_c}, the set of communication events shared by at least one other component, which are the only shared events between components. Figure 1 depicts a self-observed distributed system with two components: G_1 (left) and G_2 (right), where the events Oi denote locally observable events, the event F denotes an unobservable fault event, the events Ui denote unobservable normal events and the events Ci denote communication events.

We denote the synchronized FSM of components $G_1, ..., G_n$ by $\|(G_1, ..., G_n)$, where the synchronized events are the shared events between components and any one of them

 ISBN: 978-1-61208-233-2

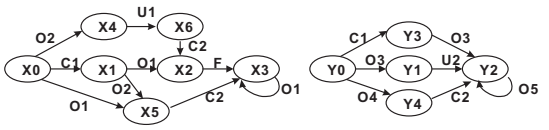


Fig. 1. A system with two components G_1 (left) and G_2 (right).

always occurs simultaneously in all components that define it. The global model of the entire system is implicitly defined as the synchronized FSM of all components based on their shared events, i.e., communication events. However, the global model will not be calculated since in a self-observed distributed system, the global occurrence order of observable events is not accessible. In the following, we call subsystem of G the synchronization of a subset of components of G, i.e., $\|(G_{s_1}, ..., G_{s_m})$, where $\{s_1, ...s_m\} \subseteq \{1, ...n\}$. One component or the entire system can be considered as a subsystem.

Given a system model $G = (Q, \Sigma, \delta, q^0)$, the set of words produced by the FSM G is a prefix-closed language $L(G)$ that describes the normal and faulty behaviors of the system. Formally, $L(G) = \{s \in \Sigma^* | \exists q \in Q, (q^0, s, q) \in \delta\}$, where the transition δ has been extended from events to words. In the following, we call a word of $L(G)$ also a **trajectory** in the system G and a sequence $q_0\sigma_0q_1\sigma_1...$ a **path** in G, where $\sigma_0\sigma_1...$ is a trajectory and for all i, we have $(q_i, \sigma_i, q_{i+1}) \in \delta$. Given $s \in L(G)$, we denote the post-language of $L(G)$ after s by $L(G)/s$ and denote the projection of s to observable events of G (resp. G_i) by $P(s)$ (resp. $P_i(s)$). We adopt the assumption in [3], i.e., the projection of the global language on each local model is observable live, i.e., there is no unobservable cycle in any component. For the sake of simplicity, our approach is shown for only one fault, which can be extended to the case with multiple faults by running one time for each fault. Next we rephrase reconstructibility introduced in [7].

Definition 1: (Reconstructibility). Given a system G that is composed of several subsystems, i.e., $G = \|(G_{s_1}, ..., G_{s_m})$, a set of trajectories in these subsystems is said to be reconstructible with respect to G if it is obtained by projection on this set of subsystems of a trajectory in G.

If there is no common communication event between two subsystems, then any trajectory in one subsystem and any one in the other subsystem are reconstructible.

For the sake of consistency, now we rename what is called cooperative diagnosability in [5] as joint diagnosability. We denote a trajectory ending with the fault f by s^f.

Definition 2: (Joint diagnosability). A fault f is jointly diagnosable in a self-observed distributed system G with components $\{G_1, ...G_n\}$, iff

$$\exists k \in N, \forall s^f \in L(G), \forall t \in L(G)/s^f, (\forall i \in \{1, ..., n\}, |P_i(t)| \geq k) \Rightarrow (\forall p \in L(G) (\forall i \in \{1, ..., n\}, P_i(p) = P_i(s^f.t)) \Rightarrow f \in p).$$

Joint diagnosability of f means that for each faulty trajectory s^f in G, for each extension t with enough locally observable events in all components, every trajectory p in G that is equivalent to $s^f.t$ for local observations in each component should contain in it f. In other words, the fault can be detected after finite non bounded trajectory prolongation in at least one component. In a self-observed system, we call a pair of trajectories p and $p\prime$ satisfying the three conditions a (global) **indeterminate pair**: 1) p contains f and $p\prime$ does not; 2) p has arbitrarily long local observations in all components after the occurrence of f; 3) $\forall i \in \{1, ..., n\}$, $P_i(p) = P_i(p\prime)$. Here arbitrarily long local observations can be considered as infinite local observations. Now we have the following theorem [5].

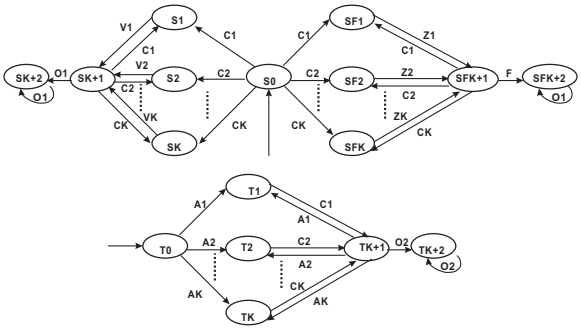


Fig. 2. A system with two components G_1 (top) and G_2 (bottom) for proving undecidability of joint diagnosability.

Theorem 1: Given a self-observed distributed system G, a fault f is jointly diagnosable in G iff there is no (global) indeterminate pair in G.

III. UNDECIDABLE CASE

To discuss about joint diagnosability, we consider two cases: communication events being unobservable and observable. We first consider the general case, i.e., communication events being unobservable.

Theorem 1 implies that checking joint diagnosability boils down to check the existence of indeterminate pairs that witnesses non joint diagnosability. Inspired from [6], where undecidability of joint observability is proved by reducing the Post's Correspondence Problem (PCP) to an observation problem, we discuss first about whether joint diagnosability is decidable or not.

For the sake of simplicity, we give now a simplified proof for undecidability of joint diagnosability.

Theorem 2: Given a self-observed distributed system where communication events are unobservable, checking joint diagnosability of a given fault is undecidable.

Proof:

1) PCP: given a finite alphabet Σ, two sets of words $v_1, v_2, ..., v_k$ and $z_1, z_2, ..., z_k$ over Σ, then a solution to PCP is a sequence of indices $(i_m)_{1 \leq m \leq n}$ with $n \geq 1$ and $1 \leq i_m \leq k$ for all m such that $v_{i_1}v_{i_2}...v_{i_n} = z_{i_1}z_{i_2}...z_{i_n}$.

2) Now consider the example depicted in Figure 2, where the system is composed of two components G_1 and G_2. In G_1, each one of $Vi, i \in \{1, ..., k\}$, and each one of $Zi, i \in \{1, ..., k\}$, denotes a sequence of observable events all different from $O1$, $C1, ..., Ck$ are unobservable communication events, F denotes a fault event and $O1$ is an observable event. In G_2, each one of $Ai, i \in \{1, ..., k\}$, denotes an observable event different from $O2$, $C1, ..., Ck$ are unobservable communication events and $O2$ is an observable event. Then the observations in G_1 can be described as $Vi_1Vi_2...Vi_nO1^*$ without fault or

$Zi_1Zi_2...Zi_nO1^*$ with fault, where $\forall i_j, j \in \{1,...,n\}, i_j \in \{1,...,k\}$. In G_2, the observations are $Ai_1Ai_2...Ai_nO2^*$. In this system, the occurrence of the fault can be confirmed by the observation of $O1$.

3) Without the observation of $O1$, the local observations are $wO1^+$ for G_1 and $Ai_1Ai_2...Ai_nO2^*$ for G_2, where $w = Vi_1Vi_2...Vi_n$ when there is no fault or $w = Zi_1Zi_2...Zi_n$ when there is a fault. Clearly, if PCP has a solution, i.e., $\exists(i_m)_{1\leq m\leq n}$ such that $Vi_1Vi_2...Vi_n = Zi_1Zi_2...Zi_n$, we have two trajectories p and $p\prime$ such that the observations of p in G_1 are $Vi_1Vi_2...Vi_nO1^+$, which is a trajectory without fault, while the observations of $p\prime$ in G_1 are $Zi_1Zi_2...Zi_nO1^+$, which is a trajectory with a fault. And both p and $p\prime$ have the same observations for G_2, i.e., $Ai_1Ai_2...Ai_nO2^*$. Thus we get that p and $p\prime$ have the same observations for both G_1 and G_2, i.e., $Vi_1Vi_2...Vi_nO1^+$=$Zi_1Zi_2...Zi_nO1^+$ for G_1 and $Ai_1Ai_2...Ai_nO2^*$ for G_2, then the fault is not jointly diagnosable.

4) On the other hand, if the fault is not jointly diagnosable, then we obtain at least one indeterminate pair, denoted by p and $p\prime$ such that the projection of p on G_1 is $Ci_1Vi_1Ci_2Vi_2...Ci_nVi_nO1^*$, on G_2 is $Ai_1Ci_1Ai_2Ci_2...Ai_nCi_nO2^*$ and that of $p\prime$ on G_1 is $Cj_1Zj_1Cj_2Zj_2...Cj_mZj_mFO1^*$ and on G_2 is $Aj_1Cj_1Aj_2Cj_2...Aj_mCj_mO2^*$. From the fact that p and $p\prime$ have the same observations for G_2, we get $Ai_1Ai_2...Ai_nO2^* = Aj_1Aj_2...Aj_mO2^*$ and thus we have $m = n$ and $i_1 = j_1, ..., i_n = j_n$. And then from the same observations of p and $p\prime$ on G_1, we get $Vi_1Vi_2...Vi_nO1^* = Zi_1Zi_2...Zi_nO1^*$, i.e., $Vi_1Vi_2...Vi_n = Zi_1Zi_2...Zi_n$, which means that there is a solution for PCP.

5) The above proves that the existence of a solution for PCP is equivalent to that of a fault being not jointly diagnosable. Since PCP is an undecidable problem, then checking joint diagnosability is undecidable. ■

There are two major differences between joint diagnosability in our framework and joint observability in [6]. One is that the former assumes that local observers are attached to local components that are synchronized by common communication events to get a global model while the latter separates arbitrarily the observable events in the global model into several sets. The other one is that joint diagnosability consists in separating infinite trajectories while joint observability consists in separating finite ones. Thus, if any communication event is assumed to be unobservable, joint diagnosability checking boils down to infinite PCP. But this one has also been proved to be undecidable [8], which gives the result.

IV. SUFFICIENT ALGORITHM

We have proved that joint diagnosability with unobservable communication events is undecidable. We can nevertheless propose an algorithm to test a sufficient condition, which is still quite useful in some circumstances. We first construct the local diagnoser from a given local model to show fault information for any local trajectory. Then, we show how to build a structure called local twin plant to obtain original information about indeterminate pairs (also called local indeterminate pairs in the following), based on the local diagnoser. The next step is to check the global consistency, i.e., to check whether the local indeterminate pairs can be extended into (global) indeterminate pairs, whose existence testifies non joint diagnosability. Actually, our algorithm remains trivially applicable when the assumption of unobservability of communication events is partially relaxed, i.e., in the most general case where some communication events are observable and others unobservable.

A. Original diagnosability information

To check the existence of indeterminate pairs, in the distributed framework, we use the structure called local twin plant defined in [2]. In particular, the considered fault is assumed to only occur in one component, denoted by G_f. Then the local twin plant for G_f contains original information of indeterminate pairs: actually this twin plant is a FSM that compares every pair of local trajectories to search for the pairs with the same arbitrarily long local observations, but exactly one of the two containing a fault, which are local indeterminate pairs. First, we define delay closure operation with respect to a subset Σ_d of Σ to preserve only the information about the events in Σ_d.

Definition 3: (Delay Closure). Given a FSM G = (Q, Σ, δ, q^0), its delay closure with respect to $\Sigma_d \subseteq \Sigma$ is $\complement_{\Sigma_d}(G) = (Q, \Sigma_d, \delta_d, q^0)$ where $(q, \sigma, q') \in \delta_d$ iff $\exists s \in (\Sigma \backslash \Sigma_d)^*, (q, s\sigma, q') \in \delta$.

We now describe how to construct the local diagnoser of a given component, based on which we build the local twin plant. Given a local model, we get a modified one by attaching fault label, denoted by $l \in \{N, F\}$, where N for normal and F for fault, to each state. In other words, before the occurrence of the fault, each state is labeled with label N and, after its occurrence, with label F.

Definition 4: (Local diagnoser). Given a local model G_i, its local diagnoser D_i is obtained by operating the delay closure with respect to the set of communication events and observable events on the modified model: $D_i = \complement_{\Sigma_{i_o} \cup \Sigma_{i_c}}(G_i^m)$, where G_i^m is the modified version of G_i.

Based on the local diagnoser, the corresponding local twin plant is obtained by synchronizing the local diagnoser with itself based on the locally observable events, allowing one to obtain all pairs of local trajectories with the same observations to search for local indeterminate pairs. To simplify this synchronization, the two identical local diagnosers, denoted by D_i^l (left instance) and D_i^r (right instance), can be reduced as follows: D_i^l is obtained by retaining only paths with at least one fault cycle and D_i^r is obtained by retaining only paths with at least one non-fault cycle. This reduction keeps all original diagnosability information since what we are interested in are only local indeterminate pairs. However, this reduction is only applicable for the local diagnoser of the faulty component G_f; for other components, the local twin plant is obtained by synchronizing the non reduced left instance and the non reduced right instance since there is no fault information. Since this synchronization is based on observable events Σ_{i_o}, the

 ISBN: 978-1-61208-233-2

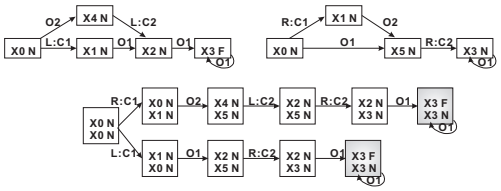


Fig. 3. Two reduced instances of the diagnoser for G_1 (top) and part of the corresponding local twin plant (bottom).

non-synchronized events are distinguished by the prefix L or R: in D_i^l (D_i^r), each communication event $c \in \Sigma_{i_c}$ from D_i is renamed by $L : c$ ($R : c$). The names of all locally observable events are left unchanged.

Definition 5: (Local twin plant). Given a local diagnoser D_i for the component G_i, the corresponding local twin plant is a FSM, denoted by $T_i = D_i^l \| D_i^r$, where the synchronized events are locally observable events in G_i.

Each state of a local twin plant is a pair of local diagnoser states providing two possible diagnoses with the same local observations. Given a twin plant state $((q^l, l^l)(q^r, l^r))$, if the considered fault $f \in l^l \cup l^r$ but $f \notin l^l \cap l^r$, which means that the occurrence of f is not certain up to this state, then this state is called an ambiguous state with respect to the fault f. An ambiguous state cycle is a cycle containing only ambiguous states. In a local twin plant, if a path contains an ambiguous state cycle with at least one locally observable event, then it is called a **local indeterminate path**, which corresponds to a local indeterminate pair. Note that local indeterminate paths contain original diagnosability information and can be obtained only in the local twin plant of the component G_f. If a local indeterminate pair can be extended into a global indeterminate pair, then we say that its corresponding local indeterminate path is globally consistent. Figure 3 shows the left and right instances of the local diagnoser for the faulty component G_1 of Figure 1 (top) as well as a part of the corresponding local twin plant (bottom). Clearly, in the local twin plant, we have local indeterminate paths since they have ambiguous state cycles with observable events.

B. Global consistency checking

Joint diagnosability verification consists in checking the existence of globally consistent local indeterminate paths, whose existence proves non joint diagnosability. To do this, we have to check the global consistency of the corresponding left trajectories of the local indeterminate paths in the local twin plants as well as that of their corresponding right trajectories, shortly called left consistency checking and right consistency checking.

Definition 6: (Left (Right) consistent plant). Given a subsystem G_S composed of components $G_{i_1}, ..., G_{i_m}$ and their corresponding local twin plants $T_{i_1}, ..., T_{i_m}$, to obtain a left (right) consistent plant with respect to the subsystem G_S, denoted by T_f^l (T_f^r), we perform the following two steps:

1) Distinguish right (left) communication events between local twin plants by renaming them with the prefix of component ID. For example, $R{:}C2$ ($L{:}C2$) in the local twin plant of G_2 is renamed as $G_2{:}R{:}C2$ ($G_2{:}L{:}C2$).

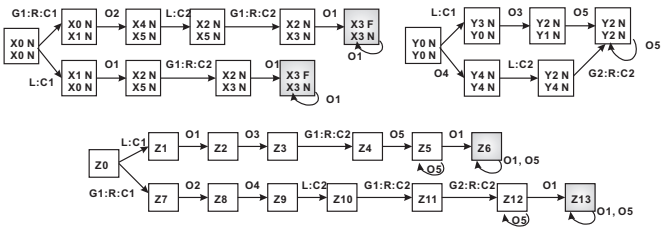


Fig. 4. Part of the renamed local twin plants for G_1 and G_2 (top) and part of the left consistent plant T_f^l (bottom).

2) Synchronize the renamed local twin plants with the synchronized events being the common left (right) communication events, which works because observable events do not intersect between components and non-synchronized right (left) communication events are distinguished by the prefix of component ID.

From definition 1, we know that in the left (right) consistent plant with respect to a subsystem G_S, each path p corresponds to a set of paths $p_{i_1}, ..., p_{i_m}$ in the local twin plants of all components in G_S such that the set of left (right) trajectories of $p_{i_1}, ..., p_{i_m}$ are reconstructible with respect to G_S. For our example, the bottom part of Figure 4 shows a part of the left consistent plant T_f^l, which is obtained by synchronizing the renamed local twin plant of G_1 and that of G_2 (top part of Figure 4) based on the common left communication events.

C. Algorithm

Algorithm 1 presents the procedure to verify a sufficient condition of joint diagnosability. As shown in the pseudo-code, algorithm 1 performs as follows. Given the input as the set of component models, the fault F that may occur in the component G_f, we initialize the parameters as empty, i.e., G_S^l (G_S^r), the subsystem for the left (right) consistency checking. The procedure of the algorithm can be separated into two parts: left consistency checking (line 3-12) and right consistency checking (line 13-24).

Left consistency checking begins with the local twin plant construction of G_f, the subsystem G_S^l being now G_f (line 3-4). When both the left consistent plant T_f^l with respect to the current left subsystem G_S^l and $DirectCC(G, G_S^l)$ are not empty (line 5), where $DirectCC(G, G_S^l)$ is the set of directly connected components to G_S^l (a directly connected component being one sharing at least one common communication event with the subsystem), the algorithm repeatedly performs the following steps to further check left consistency.

1) Select one directly connected component G_i to the subsystem G_S^l and construct its local twin plant T_i (line 6-7).

2) Synchronize T_f^l with T_i to obtain left consistent plant for this extended subsystem based on common left communication events (line 8). To do this, non-synchronized right communication events are distinguished by the prefix of component ID.

3) Update the subsystem G_S^l by adding G_i and reduce the newly obtained T_f^l by retaining only paths with ambiguous state cycles containing observable events for all components in G_S^l (line 9-10).

Algorithm 1 Sufficient algorithm

1: INPUT: the system model $G = (G_1, ..., G_n)$; the fault F and the faulty component G_f
2: Initializations: $G_S^l \leftarrow \emptyset$ (subsystem for left consistency checking); $G_S^r \leftarrow \emptyset$ (subsystem for right consistency checking)
3: $T_f^l \leftarrow ConstructLTP(G_f)$
4: $G_S^l \leftarrow G_f$
5: **while** $T_f^l \neq \emptyset$ and $DirectCC(G, G_S^l) \neq \emptyset$ **do**
6: $G_i \leftarrow SelectDirectCC(G, G_S^l)$
7: $T_i \leftarrow ConstructLTP(G_i)$
8: $T_f^l \leftarrow T_f^l \| T_i$
9: $G_S^l \leftarrow Add(G_S^l, G_i)$
10: $T_f^l \leftarrow RetainConsisPaths(T_f^l)$
11: **if** $T_f^l = \emptyset$ **then**
12: return "F is jointly diagnosable in G"
13: **else**
14: $T_f^r \leftarrow AbstractRight(G_f, T_f^l)$
15: $G_S^r \leftarrow G_f$
16: **while** $T_f^r \neq \emptyset$ and $G_S^l \neq G_S^r$ **do**
17: $G_i \leftarrow SelectDirectCC(G_S^l, G_S^r)$
18: $T_f^r \leftarrow T_f^r \| AbstractRight(G_i, T_f^l)$
19: $G_S^r \leftarrow Add(G_S^r, G_i)$
20: $T_f^r \leftarrow RetainConsisPaths(T_f^r)$
21: **if** $T_f^r = \emptyset$ **then**
22: return "F is jointly diagnosable in G"
23: **else**
24: return "Joint diagnosability cannot be determined"

If the left consistent plant T_f^l is empty, then there is no local indeterminate path that corresponds to a set of paths in the local twin plants of all components in the subsystem such that their left trajectories are reconstructible (definition 1), which implies the non existence of a globally consistent local indeterminate path. In this case joint diagnosability information is returned (line 11-12). Otherwise, if T_f^l is not empty (line 13), then we proceed to check right consistency of the corresponding paths in T_f^l that have been already verified to be left consistent in the whole system.

Right consistency checking begins with the function $AbstractRight(G_f, T_f^l)$ (line 14), which performs delay closure with respect to right communication events and observable events of G_f. Then the subsystem G_S^r is assigned as G_f (line 15). When the right consistent plant T_f^r for the current right subsystem G_S^r is not empty and $G_S^l \neq G_S^r$ (line 16), we repeatedly perform the following steps to check right consistency in an extended subsystem (since left consistency checking does explore all connected components, for right consistency checking we only consider the subsystem G_S^l instead of the whole system).
1) Select a directly connected component G_i to G_S^r from G_S^l (line 17).
2) Perform the function $AbstractRight(G_i, T_f^l)$, which has been described as above, and then synchronize with T_f^r based on the common right communication events (line 18). To do this, we rename the right communication events by removing the prefix of component ID, e.g., G_i:R:$C2$ renamed as R:$C2$. 3) Update the subsystem G_S^r by adding G_i and reduce the newly obtained T_f^r by retaining only paths with ambiguous state cycles containing observable events for all components in G_S^r (line 19-20).

If the right consistent plant T_f^r is empty, then there is no local indeterminate path that corresponds to a set of paths in the local twin plants such that their left trajectories and right trajectories are reconstructible respectively, i.e., there is no globally consistent local indeterminate path. In this case, the algorithm returns joint diagnosability information (line 21-22). Otherwise, if T_f^r is not empty, we cannot determine whether the fault is jointly diagnosable or not. Then the algorithm returns indetermination information (line 23-24). In other words, empty left consistent plant T_f^l or empty right consistent plant T_f^r is a sufficient condition but not a necessary condition of joint diagnosability.

Theorem 3: In algorithm 1, if the left consistent plant T_f^l or the right consistent plant T_f^r is empty, then the fault is jointly diagnosable, but the reverse is not true.

Proof: :
($\Rightarrow$) Suppose that T_f^l or T_f^r is empty and that the fault is not jointly diagnosable. From non joint diagnosability, it follows that there exists at least one globally consistent local indeterminate path. Since global consistency of a local indeterminate path implies both left consistency and right consistency, from algorithm 1 we know that, after left and right consistency checking, this local indeterminate path must correspond to a path both in T_f^l and in T_f^r. Thus neither T_f^l nor T_f^r is empty, which contradicts the assumption.
($\Leftarrow$) Now we explain why non emptiness of both T_f^l and T_f^r does not necessarily imply that the fault is not jointly diagnosable. Suppose that T_f^l is not empty and that it contains two paths, denoted by $\rho 1$ and $\rho 2$, corresponding to two local indeterminate paths. $\rho 1$ corresponds to a set of paths $\rho_i^1, 1 \leq i \leq n$ in the local twin plants of all components and $\rho 2$ corresponds to a set of paths $\rho_i^2, 1 \leq i \leq n$ in all local twin plants. Now suppose that the right trajectories of the set of paths $\rho_i^1, 1 \leq i \leq n$ are not reconstructible and the same for that of the set of paths $\rho_i^2, 1 \leq i \leq n$. It follows that the two local indeterminate paths cannot be extended into global indeterminate pairs and thus are not globally consistent. Then we further suppose that the right trajectories of the set of paths $\rho_1^1, ..., \rho_{n-1}^1, \rho_n^2$ are reconstructible or the same for the set of paths $\rho_1^2, ..., \rho_{n-1}^2, \rho_n^1$. In this case, from algorithm 1, it follows that finally the right consistent plant T_f^r is not empty. Now both T_f^l and T_f^r are not empty but there is no globally consistent local indeterminate paths, i.e., the fault is jointly diagnosable. ■

Now, we illustrate on our example the fact that the condition is not necessary. The top part of Figure 5 shows the results of performing delay closure with respect to right communication events and observable events both for

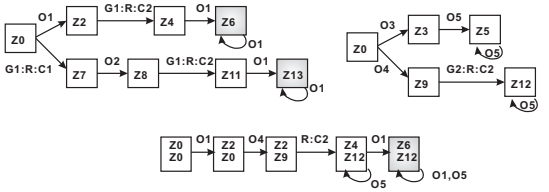


Fig. 5. FSM after delay closure on the left consistent plant (Figure 4) for G_1 and G_2 (top) and part of the right consistent plant (bottom).

G_1 and G_2 on the left consistent plant depicted in Figure 4. Then, to check right consistency, we rename again the right communication events by removing the component ID such that they can be synchronized. The bottom part of Figure 5 shows a part of the right consistent plant, which is not empty. Now, both left and right consistent plants are not empty, but this does not imply the existence of global indeterminate pairs that witnesses non joint diagnosability. Actually, the part of the left consistent plant depicted here corresponds to two local indeterminate pairs in G_1 with their corresponding left consistent pairs in G_2, i.e., one local indeterminate pair is $((C1.O1.F.O1^*), (O1.C2.O1^*))$ in G_1 with its left consistent pair $((C1.O3.O5^*), (O3.U2.O5^*))$ in G_2 and the other local indeterminate pair is $((O2.U1.C2.F.O1^*), (C1.O2.C2.O1^*))$ in G_1 with its left consistent pair $((O4.C2.O5^*), (O4.C2.O5^*))$ in G_2. While the right consistent plant shown here corresponds to one local indeterminate pair in G_1, which is $((C1.O1.F.O1^*), (O1.C2.O1^*))$, with its right consistent pair in G_2, i.e., $((O4.C2.O5^*), (O4.C2.O5^*))$. Thus, we can see that the same local indeterminate pair does not correspond to the same consistent pair in G_2 in the left consistent plant and in the right consistent plant, which means that this local indeterminate pair cannot be extended into a global indeterminate pair. Our algorithm gives indeterminate information for joint diagnosable systems that satisfy the following condition: for any set of paths including one path in the local twin plant of each non faulty component and one local indeterminate path in that for faulty component, if they are left consistent and right consistent respectively, then their corresponding local trajectories in the components cannot constitute an indeterminate pair through synchronization. Our illustrated example is quite tricky to show the possibility of indeterminate decision given by our algorithm for a joint diagnosable system. However, in reality, a system satisfying the above condition is quite rare and thus this algorithm can be applicable for a large number of complex systems.

V. DECIDABLE CASE

We have proved the undecidability of joint diagnosability with unobservable communication events. If we assume their observability, then this problem becomes decidable. Because when any communication event is observable, in the local twin plant, we obtain all pairs of local trajectories with the same observations, including the same observable communication events. Thus, each path in the local twin plant corresponds to a pair of local trajectories with the same sequence of communication events. It follows that, during global consistency checking, the separate checkings for left and right consistency becomes now only one checking. While in algorithm 1, the checking into two separate phases is the reason why it gives only a sufficient but not necessary condition. Actually, the observability of communication events makes joint diagnosability equivalent to classical diagnosability since only one checking for global consistency implies the same global occurrence order of observations for global indeterminate pairs.

VI. CONCLUSION AND FUTURE WORK

In this paper, we consider self-observed distributed systems where observable events can only be observed by their own component and thus the distributed and private (w.r.t. observation) nature of real systems is taken into account. Then, we prove the undecidability of joint diagnosability checking when communication events are unobservable, before proposing an algorithm to test a sufficient condition. We start from local indeterminate paths and then we check both in sequence left consistency and right consistency. Due to the observation-privacy, the global occurrence order of observable events between different components is not known, which is taken into account through constructing left and right consistent plants separately. For computational complexity, as distributed diagnosability approaches with globally observable events, in the worst case, our algorithm has polynomial complexity in the number of system states and exponential complexity in the number of system components. But our approach is more autonomous thanks to distributed observations. Then we briefly discuss the decidable case where communication events are observable. There is a gap between these two cases as the unobservable case is undecidable and the observable case is decidable. Next interesting work is to investigate where is the frontier between the two cases, i.e., to study the decidability of joint diagnosability for partial observability of communication events.

REFERENCES

[1] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, "Diagnosability of discrete event systems," in *IEEE Transactions on Automatic Control*, 1995, pp. 40(9):1555–1575.

[2] S. Jiang, Z. Huang, V. Chandra, and R. Kumar, "A polynomial time algorithm for diagnosability of discrete event systems," in *IEEE Transactions on Automatic Control*, 2001, pp. 46(8):1318–1321.

[3] Y. Pencolé, "Diagnosability analysis of distributed discrete event systems," in *Proceedings of 16th European Conference on Artificial Intelligence ECAI'04*, Valencia, Spain, August 2004, pp. 43–47.

[4] A. Schumann and Y. Pencolé, "Scalable diagnosability checking of event-driven systems," in *Proceedings of 20th International Joint Conference on Artificial Intelligence IJCAI-07*, Hyderabad, India, 2007, pp. 575–580.

[5] L. Ye and P. Dague, "Diagnosability analysis of discrete event systems with autonomous components," in *Proceedings of 19th European Conference on Artificial Intelligence ECAI-10*, Lisbon, Portugal, August 2010, pp. 105–110.

[6] S. Tripakis, "Undecidable problems of decentralized observation and control," in *40th IEEE Conference on Decision and Control*, Orlando, Florida, December.

[7] R. Cori and Y. Métivier, "Recognizable subsets of some partially abelian monoids," *Theoretical Computer Science*, vol. 35, pp. 179–189, 1985.

[8] V. Halava and T. Harju, "Undecidability of infinite post correspondence problem for instances of size 9," *Theoretical Informatics and Applications*, vol. 40, pp. 551–557, 2006.

 ISBN: 978-1-61208-233-2

A Combined Formal Analysis Methodology and Towards Its Application to Hierarachical State Transition Matrix Designs

Weiqiang Kong
Graduate School of IS&EE, Kyushu University, Japan.
weiqiang@qito.kyushu-u.ac.jp

Leyuan Liu
Graduate School of IS&EE, Kyushu University, Japan.
leyuan@f.ait.kyushu-u.ac.jp

Hirokazu Yatsu
Graduate School of IS&EE, Kyushu University, Japan.
hirokazu.yatsu@f.ait.kyushu-u.ac.jp

Akira Fukuda
Graduate School of IS&EE, Kyushu University, Japan.
fukuda@ait.kyushu-u.ac.jp

***Abstract*—Interactive theorem proving and model checking are known as two formal verification techniques that have complementary features and aims, but overlapping application areas. In this paper, we investigate a procedure (methodology) called Combined Falsification and Verification (CFV), by which the benefits of both interactive theorem proving and model checking could be enjoyed for formal analysis of software systems against invariant properties. We have been developing a SMT-based Bounded Model Checker called Garakabu2 for falsification of HSTM designs. Interfaces necessary for enabling the procedure CFV is planned to be introduced into Garakabu2 for providing an auxiliary functionality for users of Garakabu2 who are experts in formal methods.**

***Keywords*-Interactive theorem proving; Bounded Model Checking; Invariant Properties; State Transition Matrix.**

I. INTRODUCTION

As software systems grow in scale and functionality, there is an increasing demand that these systems should be reliable. This is especially the case for those systems that are safety-critical ones, such as banking systems, railway systems and aircraft guidance systems, in which subtle errors can cause fatal losses in economy and lives. One way of improving reliability of software systems is by using formal verification, which are mathematically-based techniques for specifying and verifying systems.

Interactive theorem proving [1] and model checking [2] are known as two formal-verification techniques that have complementary features and aims, but overlapping application areas. The main different characteristics between them lie with the aspects of state space (infinite vs. finite), automation (limited vs. fully), and counterexample (not automatic vs. automatic). There are no hard-and-fast answers to the priority of one to the other. It is a common consensus that the two techniques are equally effective and maintainable on the whole for complex applications, while each technique has specific strengths and weaknesses. By combining them, attempts have been made to enjoy the best of both worlds, but no comprehensive understanding exists.

In this paper, we pursue a better understanding about the combination of the two techniques through a specific combination that the OTS/CafeOBJ method [3] is backended with Maude model checkers [4]. Specifically, we focus on how the counterexamples automatically generated by model checkers can help the interactive inductive verification technique of theorem proving for invariant properties.

Particularly, regarding how to combine the two techniques, we have previously proposed a procedure called Induction-Guided Falsification (IGF) in [5]. IGF is a procedure that can reveal logical errors (i.e., falsification) lurking in the specifications for theorem proving as early as possible by employing model checking during the interactive inductive verification of invariant properties. As an extension of IGF with respect to verification, we investigate a procedure called Combined Falsification and Verification (CFV). CFV is a more general procedure that combines interactive inductive verification of invariant properties with model checking, which is supposed to be followed by human verifiers.

We have been developing a SMT-based [6] Bounded Model Checking (BMC) [7] tool called Garakabu2 [8], [9] for formal analysis of designs specified in Hierarchical State Transition Matrix (HSTM) [10], a set of State Transition Matrices organized in a hierarchical structure. HSTM has been widely accepted and used by particularly Japanese embedded software industry, and been adopted as the modeling language of commercial model-based tools. However, one issue is that Garakabu2 only conducts falsification due to that only a bounded state space is checked with BMC. We plan to implemented interfaces necessary for enabling the procedure CFV, and thus, make Garakabu2 usable for conducting formal analysis of HSTM designs with both interactive theorem proving and model checking.

The paper is organized as follows. Section II introduces preliminary knowledge. Section III first reviews IGF and then proposes the procedure CFV. Section IV investigates briefly the feasibility/possibility of introducing CFV into Garakabu2, and Section V concludes the paper.

II. PRELIMINARY

The procedures IGF and CFV are proposed and described based on the combination of the OTS/CafeOBJ method (for interactive theorem proving) and Maude Model Checkers. In this section, we informally review these two methods/techniques, while refer readers to [3] and [4] for

 ISBN: 978-1-61208-233-2

their respective formal details. We use a simple mutual exclusion algorithm using a queue to demonstrate how to use the OTS/CafeOBJ method, and respectively, the Maude LTL model checker to specify and verify invariant properties. The pseudo-code executed by each process i repeatedly can be described as follows:

$l1:$ $put(queue, i)$;
$l2:$ **repeat until** $top(queue) = i$
critical section;
$cs:$ $get(queue)$;

$queue$ is the queue of process IDs shared by all processes. $put(queue, i)$ puts a process ID *i* into *queue* at the end, $get(queue)$ deletes the top element from $queue$, and $top(queue)$ returns the top element of $queue$. Each iteration of the loop at label l2 is supposed to be atomically processed. Initially each process i is at label $l1$ and $queue$ is *empty*.

A. The OTS/CafeOBJ Method

The OTS/CafeOBJ method [3] is a modeling, specification and verification method. In the OTS/CafeOBJ method, a system to be verified is first modeled as an observational transition system (OTS), a transition system that can be straightforwardly written in terms of equations. The OTS is then written in CafeOBJ [11] as a behavioral specification (Some basic data types used in the OTS, such as `Nat` and `Int` are described as general algebraic specifications, which are imported in this behavioral specification).

The OTS/CafeOBJ specification of the sample mutual exclusion algorithm consists of three data type modules (with the names `LABEL`, `PID` and `QUEUE`) and one OTS module (with the name `QLOCK`). The three data type modules define sorts `Label`, `Pid` and `Queue`, respectively. We show module `LABEL` as an example and the other two are defined similarly. `Label` is written in CafeOBJ as:

```
mod! LABEL {
  [Label]
  ops l1 l2 cs : -> Label
  op _=_ : Label Label -> Bool {comm}
  var L : Label
  eq (L = L) = true .     eq (l1 = l2) = false .
  eq (l1 = cs) = false .  eq (l2 = cs) = false .
}
```

In the module `LABEL`, `[Label]` is the declaration of the sort `Label`; `l1`, `l2` and `cs` are declared constants; `L` is a declared variable. Note that operator `_=_` is the equality predicate for sort `Label`.

The OTS module specifies behaviors (state transitions) of the algorithm. The sort denoting states of the OTS is declared as `Sys`. The operators denoting the observers and transitions are declared as follows (where '`--`' marks the rest of the line as a comment):

```
-- observers
bop pc    : Sys Pid -> Label
bop queue : Sys     -> Queue

-- transitions
bop want  : Sys Pid -> Sys
bop try   : Sys Pid -> Sys
bop exit  : Sys Pid -> Sys
```

`Pid`, `Label` and `Queue` are the sorts denoting process IDs, labels and queues of process IDs, respectively. The corresponding data type modules (`LABEL`, `PID` and `QUEUE`) defining these three sorts are imported in the OTS module.

Let `I`, `J` be CafeOBJ variables for `Pid`, and `S` be a CafeOBJ variable for the hidden sort `Sys` of the OTS. Operator `try` is defined with the following equations:

```
-- for try
op c-try : Sys Pid -> Bool
eq c-try(S,I)
   = (pc(S,I) = l2 and top(queue(S)) = I) .
--
ceq pc(try(S,I),J)
    = (if I = J then cs
                else pc(S,J) fi)  if c-try(S,I) .
ceq queue(try(S,I)) = queue(S)    if c-try(S,I) .
ceq try(S,I) = S                  if not c-try(S,I) .
```

`c-try(S,I)` denotes the effective condition of the transition `try`, which checks whether process `I`'s label is `l2` and the top element of the queue is equal to `I`. If the effective condition is satisfied, the transition `try` will be executed, and the execution will change the return value of observer `pc` to `cs` if the two processes `I` and `J` are the same. The execution of transition `try` does not change the return value of observer `queue`. If the effective condition does not hold, the state is not changed, which is described by the last equation. The other two operators `want` and `exit` could be defined with CafeOBJ equations in a similar way, which are not shown here.

In the OTS/CafeOBJ method, the verification of invariant properties is mainly done by structural induction, which means that what we need to do is to show firstly that the predicate to be proven invariant holds on any initial state (called the base case), and then to show that the predicate is preserved by execution of all transitions of the OTS (called the inductive case). In each inductive case, the case is usually split into multiple sub-cases with basic predicates (equations) declared in the CafeOBJ specification.

B. Maude Model Checker

Maude [4] is a high-performance language and system supporting both equational and rewriting logic computation for a wide range of applications. An important feature of Maude is that it has model checking facilities such as the `search` command and the Maude LTL model checker.

The basic units of Maude specifications are modules. There are two kinds of modules: functional modules and system modules. Maude functional modules define data types and operations on them by means of equational theories. System modules specify the initial model of a rewrite

 ISBN: 978-1-61208-233-2

theory, which are essentially transition systems. A rewrite specification has rule statements: `crl [`$Label$`]` $T_1 \Rightarrow T_2$ `if` C_1 `/\` C_2 `/\ ... /\` C_k, in addition to the contents of functional modules. The condition part can be omitted if it is `true`. For a *finite* system, Maude `search` command explores all possible execution paths from the starting term (that represents an initial state) for reachable states satisfying some property.

C. A Specification Translation Method

We have proposed in [12] a way of translating CafeOBJ specifications for OTSs (the OTS/CafeOBJ specifications) into Maude specifications of a kind of rewriting transition systems for Bounded OTSs (the RWTS/Maude specifications). Bounded OTSs are the extension of OTSs to make it possible for the model checkers to explore a finite reachable state space of an OTS for counterexamples. To express the OTS/CafeOBJ expressible invariant properties in Maude forms, we have also proposed a simple way to generate Maude `search` commands from OTS/CafeOBJ formulas for invariant properties. We have proved that the proposed way of translation is sound with respect to counterexamples, namely that for any counterexample reported by Maude model checkers for the translated RWTS/Maude specifications, there exists a corresponding one in the original OTS/CafeOBJ specifications. We refer readers to [12] for translation details.

III. The Procedure of Combined Falsification and Verification (CFV)

In this section, we first give a brief review of the procedure Induction Guided Falsification (IGF) [5], and then introduce our proposed procedure – Combined Falsification and Verification (CFV), an extension of IGF.

A. A Review of Induction Guided Falsification (IGF)

As mentioned above, in the OTS/CafeOBJ method, although some invariant properties may be proved by rewriting and/or case splitting only, the generally used verification technique for proving invariant properties is structural induction [3]. The general procedure of structural induction is that: first, checking the base case, to show whether the state predicate to be proven invariant holds on any initial state, and second, checking the inductive case, to show whether the state predicate is preserved by the execution of any transition of the system. During proving the inductive case, we may have to discover and use other state predicates (called auxiliary state predicates) to strengthen the inductive hypothesis. Finding suitable state predicates to strengthen the inductive cases may be the most critical and difficult part of formal verification using theorem proving.

Structural induction works well when a state predicate to be proven invariant is indeed an invariant. However, it is quite often that we are trying to prove some state predicates that are essentially not invariants. Following structural induction, the usual way to know that a state predicate p under proving is not an invariant, is to show that p does not hold on any initial state, or to find some auxiliary state predicate, which is needed to prove p, but does not hold on any initial state. However, to find such an auxiliary state predicate, a lot of proof efforts are usually needed to manifest the problem. Such proof efforts can be extremely painful. Thus it is preferable that there exists some way, by which finding out errors lurking in the specifications can be easier and as earlier as possible.

Induction Guided Falsification (IGF) is a procedure that can reveal logical errors lurking in the specifications for theorem proving (falsification) as early as possible by employing model checking during the inductive verification of invariant properties, and the inductive verification can be used to reduce the state space needed for model checking to search a counterexample. The key concept that IGF lies on is *necessary lemmas*, which are obtained by applying *effective case splits*.

Definition 1: Effective case splits and Necessary lemmas. Consider proving a state predicate p to be invariant (i.e., to show $p(\upsilon)$ holds in any reachable state $\upsilon \in \mathcal{R}_{\mathcal{S}}$) by structural induction on the set of all reachable states $\mathcal{R}_{\mathcal{S}}$. In an inductive case where a transition $\tau_{y_1,\ldots,y_n}$ is taken into account, basically all we have to do is to prove $P(\upsilon_c, c_{l_1}, \ldots, c_{l_\alpha}) \Rightarrow P(\tau_{c_1,\ldots,c_n}(\upsilon_c), c_{l_1}, \ldots, c_{l_\alpha})$, where υ_c is a constant denoting an arbitrary state and each c_k is a constant denoting an arbitrary value of data type D_k. We suppose that a proposition $q_1 \vee \ldots \vee q_L$ is a tautology, where each q_l is in the form $Q_l(\upsilon_c, c_1, \ldots, c_n, c_{l_1}, \ldots, c_{l_\alpha})$. The case characterized by q_l is called a sub-case with respect to the inductive case. If the truth value of $P(\upsilon_c, c_{l_1}, \ldots, c_{l_\alpha}) \Rightarrow P(\tau_{c_1,\ldots,c_n}(\upsilon_c), c_{l_1}, \ldots, c_{l_\alpha})$ can be determined assuming each q_l, then $q_1 \vee \ldots \vee q_L$ is called an *effective case split* for this inductive case. Moreover, if the truth value is false, then $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.\ \forall y_1 : D_1, \ldots, y_n : D_n, \forall x_{l_1} : D_{l_1}, \ldots, x_{l_\alpha} : D_{l_\alpha}.\ \neg Q_l(\upsilon, y_1, \ldots, y_n, x_{l_1}, \ldots, x_{l_\alpha})$ is called a *necessary lemma* of $p(\upsilon)$.

Note that this necessary lemma can surely make the inductive case `true`. If this necessary lemma is an invariant, then it means that the arbitrary state characterized by the sub-case is not reachable, and thus the false case is discharged and p is possibly an invariant; otherwise if this necessary lemma is not an invariant, then it means that the arbitrary state characterized by the sub-case is reachable, and thus p is not an invariant.

The procedure IGF is constructed based on two lemmas as its theoretical foundations. In the following, let $q(\upsilon)$ be $\forall y_1 : D_1, \ldots, y_n : D_n, \forall x_{l_1} : D_{l_1}, \ldots, x_{l_\alpha} : D_{l_\alpha}.\ \neg Q(\upsilon, y_1, \ldots, y_n, x_{l_1}, \ldots, x_{l_\alpha})$, and let q_l be $Q(\upsilon_c, c_1, \ldots, c_n, c_{l_1}, \ldots, c_{l_\alpha})$ where υ_c is a constant denoting an arbitrary state and each c_k is a constant

denoting an arbitrary value of D_k.

Lemma 1: Let $\forall v : \mathcal{R}_S.\ q(v)$ be a necessary lemma of $\forall v : \mathcal{R}_S.\ p(v)$. If there exists a counterexample $ce_q \in \mathcal{CX}_{S,q}$ and $depth(ce_q) = N$, then (1) $ce_q \in \mathcal{CX}_{S,p}$, or (2) there exists a counterexample $ce_p \in \mathcal{CX}_{S,p}$ such that $depth(ce_p) = N + 1$.

Lemma 2: If $\mathcal{CX}_{S,p}$ is not empty and $depth(ce^{min}_{S,p}) = N + 1$, then there exists a necessary lemma $\forall v : \mathcal{R}_S.q(v)$ of $\forall v : \mathcal{R}_S.p(v)$ such that $\mathcal{CX}_{S,q}$ is not empty and $depth(ce^{min}_{S,q}) = N$.

Following the theories described in the above two lemmas, especially in lemma 2, we know that if a state predicate p to be proven invariant has counterexamples, then we can surely find and systematically construct some necessary lemmas. In turn, to prove these constructed necessary lemmas, if they do hold on any initial states, it is surely that we can find and systematically construct other necessary lemmas, and so on. As with this recursive process goes on, the depths of counterexamples of these necessary lemmas decrease. And from Lemma 1, we can conclude that if counterexamples exist for some necessary lemma, then p has counterexamples. This relieves us from traversing all needed necessary lemmas until we found one does not hold on any initial state.

Definition 2: Procedure IGF.
Input: an OTS and a state predicate p to be proven invariant.
Output: *Success* or *Fail*.

1. $\mathcal{P} := \{p\}$ and $\mathcal{Q} := \emptyset$.
2. Repeat the following until $\mathcal{P} = \emptyset$.
  (a) Choose a state predicate q from $\mathcal{P}$ and $\mathcal{P} := (\mathcal{P} - \{q\})$, where $q \in min\text{-}level(\mathcal{P})$.
  (b) **Model checking** q in a finite reachable state space. If found a counterexample, terminate and return *Fail*.
  (c) Prove $\forall v_{\text{init}} : \mathcal{I}.\ q(v_{\text{init}})$. If it reduces to false, terminate and return *Fail*.
  (d) Find a set $\mathcal{G}$ of **necessary lemmas** such that $\forall v.[((\bigwedge_{g \in \mathcal{G}}\ g(v)) \wedge q(v)) \Rightarrow \forall \tau_{y_1,\ldots,y_n} q(\tau_{y_1,\ldots,y_n}(v))]$ reduces to true.
  (e) $\mathcal{Q} := \mathcal{Q} \cup \{q\}$ and $\mathcal{P} := \mathcal{P} \cup (\mathcal{G} - \mathcal{Q})$.
3. Terminate and return *Success*.

The basic idea of the procedure IGF is that: whenever trying to prove a state predicate, which is either the state predicate concerned (say p) or a constructed necessary lemma, model checking it first. Since model checking only checks a finite reachable state space, we use structural induction to prove p even if model checking did not find any counterexample. The falsifying and verifying is conducted in a breadth-first order with respect to the proof tree (to be introduced later), which is guaranteed by selecting a state predicate of minimal level in each loop described in step 2.(a).

B. The Algorithm of CFV

We have proved in [5] that IGF is sound and complete with respect to falsification, and is sound but not complete with respect to verification. This implies that IGF may work well for proving a state predicate with counterexamples, namely that for falsifying it. But in the situation that a given state predicate is indeed an invariant (no counterexample), the procedure may not terminate and successfully prove the state predicate due to using necessary lemmas as the only way to strengthen inductive hypothesis.

As an extension of IGF for enhancing the verification capability, Combined Falsification and Verification (CFV) is a more general procedure that aims at both falsification and verification. The main difference between the procedures IGF and CFV lies on using what kind of lemmas to strengthen the inductive hypothesis. In the procedure IGF, we always construct and use necessary lemmas to strengthen inductive hypothesis, but in the procedure CFV, we systematically use some other stronger lemmas (we say a state predicate p is stronger than another q if $p \Rightarrow q$) that may be more simple and appropriate, and until no such stronger lemmas suffice to strengthen inductive hypothesis, the necessary lemmas are used at last, which are the weakest lemmas.

The algorithm of the procedure CFV is shown in Definition 3. Basic idea of the procedure CFV is almost same as the procedure IGF. But since the state predicates used to strengthen the inductive hypothesis are sometimes not necessary lemmas, we need to consider more (rather than directly concluding that the state predicate concerned is not an invariant, as done in IGF) when a counterexample is reported for a state predicate, or the state predicate does not hold on any initial state, because in both cases, what we know is only that the state predicate itself is not an invariant.

The key operation or function in the procedure CFV is $process$. When a counterexample is found by model checking for a state predicate, or the state predicate does not hold on any initial state, operation $process$ is called and it returns different values according to the category of the state predicate. The possible output of operation $process$ is either F, which means the procedure CFV should be terminated and return Fail; or (X,Y), where X denotes a set of state predicates that are not appropriate or correct and should be removed from $\mathcal{P}$ and $\mathcal{Q}$, and Y denotes a set of state predicates that are possibly appropriate or correct and should be added to $\mathcal{P}$.

The primary part (except the details of the operation $process$) of the verification procedure CFV can be represented as a flow chart as shown in Figure 1.

 ISBN: 978-1-61208-233-2

We now explain the basic idea of the procedure CFV by using some examples. Assume a tree-like structure shown in Figure 2.(a) that represents the proof of a state predicate p. The tree structure is rooted, unordered, and labeled. The tree is supposed to be constructed using a breadth-first manner. The root of the tree is p, and all the other offspring nodes are constructed lemmas (state predicates) to strengthen certain inductive hypothesis for proving their respective parent nodes (state predicates), where the nodes with superscript n are necessary lemmas and those without superscript n are not necessary lemmas.

Assume that we find a counterexample for state predicate z by model checking, or z does not hold on any initial state, which means that z is not an invariant. Since z is not a necessary lemma (without the superscript n), the procedure CFV will then use a systematical way (to be introduced later) to generate and use another state predicate, say s instead of z, to strengthen an inductive hypothesis (characterized by the label l_8) to prove r_2^n, which is shown in Figure 2.(b). And the state predicate z (and also all its children nodes, if any) will be removed from $\mathcal{P}$ and $\mathcal{Q}$, and s will be added to $\mathcal{P}$.

We now assume that the state predicate z is a necessary lemma (denoted by z^n shown in Figure 3.(a)), and we know z is not an invariant by either model checking or checking any initial state, then the procedure CFV will try to find, in its parent list, the nearest state predicate to z^n that is not a necessary lemma (assume this state predicate is x), and try to generate and use other lemmas, instead of x, to strengthen the inductive hypothesis denoted by $label(x)$. Let us see the example in Figure 3.(a), since the nearest state predicate to z^n that is not a necessary lemma is q_2, then we know that q_2 should be replaced by some other state predicates. Note that since q_2 is also used to strengthen the inductive hypothesis characterized by l_4 to prove q_1^n, then the procedure will construct two lemmas, say s_1 and s_2, to strengthen the inductive hypothesis characterized by l_4 and l_2, which is shown in Figure 3.(b). The state predicate q_2 and all its recursive children nodes should be removed from $\mathcal{P}$ and $\mathcal{Q}$, and the two newly constructed state predicate s_1 and s_2 will be added to $\mathcal{P}$.

As another example, let us see Figure 4 (see below). If we find a counterexample for the state predicate z^n, or z^n does not hold on any initial state. Since z^n is a necessary lemma, and there exists a parent list of z^n, say $r_2^n\ q_2^n$, where any state predicate in this list is a necessary lemma. Then the procedure CFV is terminated and returns Fail, which means that the state predicate p to be proven invariant is not an invariant. This situation is exactly same as the procedure IGF, which is based on the theory defined in Lemma 1.

After introducing the procedure CFV, another thing left unexplained is how the procedure *systematically* constructs other state predicates when a state predicate, which is not a necessary lemma, is not appropriate, as mentioned in

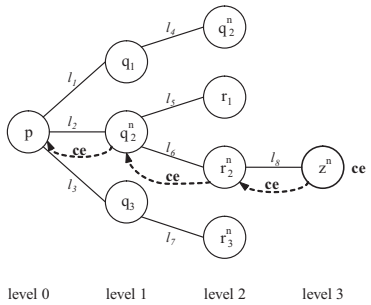


Figure 4. The third sample tree of proving p with CFV

the above examples. To explain this, let us first see more exactly what is the form of a sub-case. The sub-case is characterized by a set of equations, say E. When CafeOBJ system reduces to false for this sub-case, a necessary lemma in the form $\neg(\bigwedge_{e\in E} e)$ can be constructed. Note that from this set of equations E, we can also systematically construct other state predicates, and each such state predicate is in the form $\neg(\bigwedge_{e'\in E'} e')$, where $E' \in 2^E$, and these state predicates are stronger than the necessary lemma since $\neg(\bigwedge_{e'\in E'} e') \Rightarrow \neg(\bigwedge_{e\in E} e)$. Basically, all of these state predicates are candidates that can be used, instead of the necessary lemma, to strengthen inductive hypothesis. However, only if they satisfy two conditions, they become the real candidates that will be used by the procedure CFV. The first condition is of course they should be able to make the inductive case `true`; and the second condition is that by model checking them, no counterexample should be found.

Let us consider proving $\forall v : \mathcal{R}_{\mathcal{S}}.\ p(v)$ is an invariant. In an inductive case denoted by a transition $\tau_{y_1,\ldots,y_n}$, CafeOBJ system returns false for a sub-case characterized by a set of equations e_1, e_2, e_3, e_4. Then all the lemmas we can construct from these equations are shown below:

One equation	$\neg e_1$, $\neg e_2$, $\neg e_3$, $\neg e_4$
Two equations	$\neg(e_1 \wedge e_2)$, $\neg(e_1 \wedge e_3)$, $\neg(e_1 \wedge e_4)$ $\neg(e_2 \wedge e_3)$, $\neg(e_2 \wedge e_4)$, $\neg(e_3 \wedge e_4)$
Three equations	$\neg(e_1 \wedge e_2 \wedge e_3)$, $\neg(e_1 \wedge e_2 \wedge e_4)$ $\neg(e_1 \wedge e_3 \wedge e_4)$, $\neg(e_2 \wedge e_3 \wedge e_4)$
Four equations	$\neg(e_1 \wedge e_2 \wedge e_3 \wedge e_4)$

After the procedure CFV filtered some of them according to the two conditions, CFV will use the remaining lemmas to strengthen the inductive cases in an order from "One equation" to "Four equations", and the "Four equations" lemma is the necessary lemma.

IV. Towards Formal Analysis of HSTM Designs with the Procedure CFV

Hierarchical State Transition Matrix (HSTM) [13] is a table based modeling language for developing designs of software systems. A HSTM design, namely a design developed with HSTM, consists of multiple STMs organized in a hierarchical structure. Each STM models a component of the design in the form of a table and specifies behaviors

of the component when certain events are dispatched in certain states. A simple sample STM is shown below for demonstration purpose. The informal meaning of the STM is that: the STM has two states $S1$ and $S2$; there are two events $e1$ and $e2$ that may happen to the STM; if, for example, $e1$ is dispatched when the STM is in states $S1$, $action_1$ will be executed and then the STM switches to states $S2$. The other cells have similar meanings.

	$S1$	$S2$
$e1$	$S2$	$S1$
	$action_1$	$action_2$
$e2$	$S2$	$S1$
	$action_3$	$action_4$

HSTM has been widely accepted and used by particularly Japanese embedded software industry, and has been adopted as the modeling language of commercial model-based tools such as ZIPC [10]. However, despite of its popularity, there is still lack of mechanized formal verification supports for conducting rigorous and automatic analysis to improve reliability of HSTM designs. Based on this need, we have been developing a HSTM model checker called Garakabu2 [8], [9].

Garakabu2 implements SMT-based [6] Bounded Model Checking (BMC) [7] algorithms for verification of HSTM designs. In addition, specific considerations for its practical usabilities for non-experts in formal methods have been taken into account during its development. However, one issue is that Garakabu2 only conducts falsification due to that only a bounded state space is checked with BMC. This is sufficient for normal users like software engineers who wish to explore bugs in HSTM designs. But for expert users like those who have sufficient knowledge on inductive theorem proving, it may be desirable that verification functionality (i.e., proving correctness) is also available in Garakabu2.

Due to the fact that each STM is essentially a state transition system and a HSTM is just a set of STM organized in a hierarchical structure, it is possible that a HSTM design could be represented with OTS and thus be formally analyzed with the procedure CFV. One key issue in using CFV to analyze HSTM designs is to translate a HSTM design into an OTS (which is to be specified in CafeOBJ specification). This is not difficult since a parser for HSTM designs has been implemented in Garakabu2 and is ready to be used. We have been formalizing the translation rules from HSTM designs into OTSs and the details will be reported in another opportunity.

V. Conclusions and Future Work

In this paper, we first briefly reviewed the procedure IGF [5], and then described our proposed procedure CFV, which is an extension of IGF for both falsification and verification of systems specified in CafeOBJ specification (with OTS as the background concept). Note that although the proposed procedure CFV relies on some specific features of the OTS/CafeOBJ method and Maude model checkers, it may be revised and extended to combinations of inductive verification techniques of other theorem proving and other model checking techniques while remaining the basic idea of the procedure.

Furthermore, we simply investigated the possibility of applying the procedure CFV to formal analysis of HSTM designs. We have been formalizing translation rules from HSTM designs into OTSs. In the future, we plan to implement this translation in Garakabu2, and implement interfaces to connect Garakabu2 with CafeOBJ and Maude systems, by which Garakabu2 could be used by formal methods experts for conducting formal analysis of HSTM designs with both interactive theorem proving and model checking by following the CFV procedure.

References

[1] J. Mseguer, M. Palomino, and N. Martí-Oliet, "Equational abstractions." in *CADE*, 2003, pp. 2–16.

[2] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[3] K. Ogata and K. Futatsugi, "Proof Scores in the OTS/CafeOBJ Method," in *FMOODS 2003*, ser. LNCS, vol. 2884. Springer, 2003, pp. 170–184.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude 2.0 Manual: Version 2.1*, March 2004.

[5] K. Ogata, M. Nakano, W. Kong, and K. Futatsugi, "Induction-Guided Falsification," in *ICFEM 2006*, ser. LNCS, vol. 4260. Springer, 2006, pp. 114–131.

[6] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, *Handbook of Satisfiability*. IOS Press, 2009, vol. 185, ch. 26, pp. 825–885.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *5th TACAS*. Springer, 1999, pp. 193–207.

[8] W. Kong, N. Katahira, M. Watanabe, T. Katayama, K. Hisazumi, and A. Fukuda, "Formal verification of software designs in hierarchical state transition matrix with SMT-based bounded model checking," in *18th APSEC*. IEEE CS, 2011, pp. 81–88.

[9] W. Kong, T. Shiraishi, N. Katahira, M. Watanabe, T. Katayama, and A. Fukuda, "An SMT-based approach to bounded model checking of design in state transition matrix," *IEICE Transactions on Information and Systems*, vol. E94-D(5), pp. 946–957, 2011.

[10] CATS Co., Ltd., Japan, "ZIPC v10," www.zipc.com, [retrieved: October 2012].

[11] R. Diaconescu and K. Futatsugi, *CafeOBJ Report*, ser. AMAST Series in Computing. World Scientific, 1998, no. 6.

[12] W. Kong, K. Ogata, T. Seino, and K. Futatsugi, "A Lightweight Integration of Theorem Proving and Model Checking for System Verification," in *APSEC 2005*. IEEE CS, 2005, pp. 59–66.

[13] M. Watanabe, "Extended hierarchy state transition matrix design method," in *CATS Technical Report*, 1998.

 ISBN: 978-1-61208-233-2

Definition 3: Procedure CFV.
Input: an OTS and a state predicate p to be proven invariant.
Output: *Success* or *Fail*.

1. $\mathcal{P} := \{p\}$ and $\mathcal{Q} := \emptyset$.
2. Repeat the following until $\mathcal{P} = \emptyset$.
(1) Choose a state predicate q from $\mathcal{P}$ and $\mathcal{P} := (\mathcal{P} - \{q\})$, where $q \in min\text{-}level(\mathcal{P})$.
(2) Case [**Model checking** q in a finite reachable state space] *of:*
(a) Counterexample: *case* [$process(p, q)$] *of:*
(I) F, then terminate and returns *Fail*.
(II) (X,Y), then $\mathcal{Q} := (\mathcal{Q} - X)$, $\mathcal{P} := ((\mathcal{P} - X) \cup (Y - \mathcal{Q}))$.
(b) No counterexample, *case* [prove $\forall v_{\text{init}} : I.\ q(v_{\text{init}})$] *of:*
(I) reduces to false, *case* [$process(p, q)$] *of:*
(1°) *F*, then terminate and returns *Fail*.
(2°) *(X,Y)*, then $\mathcal{Q} := (\mathcal{Q} - X)$, $\mathcal{P} := ((\mathcal{P} - X) \cup (Y - \mathcal{Q}))$.
(II) reduces to true, then $\mathcal{G} := valid(q)$; $\mathcal{Q} := \mathcal{Q} \cup \{q\}$; $\mathcal{P} := \mathcal{P} \cup (\mathcal{G} - \mathcal{Q})$.
3. Terminate and return *Success*.

where:

process(m,n):
Input: two state predicates m and n.
Output: either *F* or a tuple *(X,Y)*, where X and Y are two sets of state predicates.

1. $X := \emptyset$ and $Y := \emptyset$.
2. Case [$n = m$] *of:*
(1) true, then terminate and return *F*.
(2) false, *case* [n is a *necessary lemma*] *of:*
(a) true, then *case* [$(parentList(n) = \emptyset) \vee (\exists\ List \in parentList(n)$, where any node in $List$ is a necessary lemma)] *of:*
(I) true, then terminate and return *F*.
(II) false, then *For each* $List \in parentList(n)$ *do*
(1°) $X := X \cup\ childrenSet(z)$, $Y := Y \cup\ tc\text{-}valid(previous(z), z)$, where z is the nearest state predicate in $List$ to n that is not a necessary lemma;
(2°) return *(X,Y)*.
(b) false, then
(I) $X := \{n\} \cup childrenSet(n)$;
(II) For all $z \in parent(n)$ *do*
(1°) $Y := Y \cup\ tc\text{-}valid(z, n)$;
(2°) return *(X,Y)*.

valid(m) $= \{\mathcal{G} \mid \forall v : \Upsilon.[((\bigwedge_{g \in \mathcal{G}}\ g(v)) \wedge m(v)) \Rightarrow \forall \tau_{y_1,\ldots,y_n} : \mathcal{T}.\ m(\tau_{y_1,\ldots,y_n}(v))]\}$.

tc-valid(m,n) $= n'$, where under the case denoted by $label(n)$, which includes the inductive case denoted by a transition $\tau_{y_1,\ldots,y_n} := lt(label(n))$, and a sub-case denoted by $lc(label(n))$, such that: $\forall v : \Upsilon.[n'(v) \wedge m(v) \Rightarrow m(\tau_{y_1,\ldots,y_n}(v))]$ reduces to true.

 ISBN: 978-1-61208-233-2

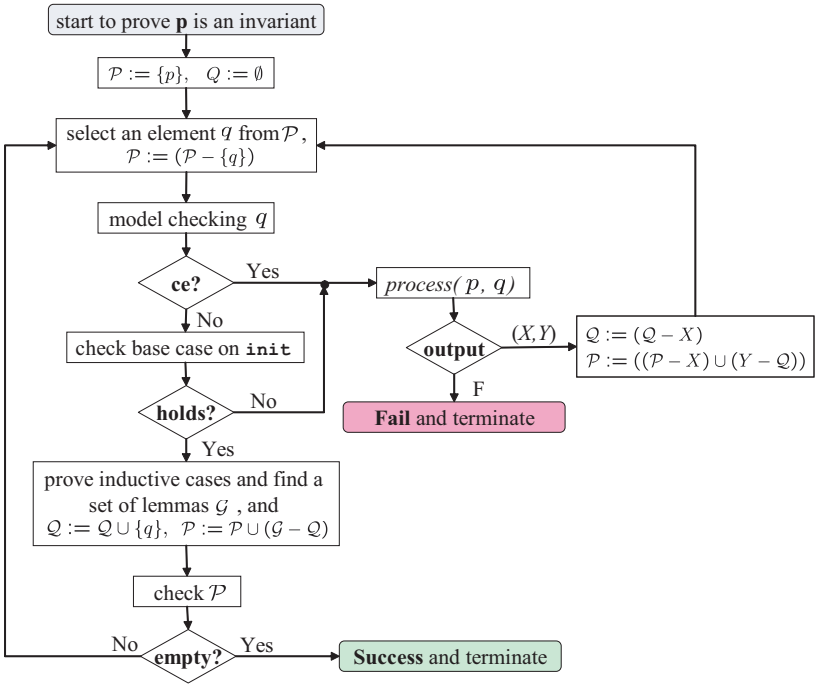


Figure 1. Flow chart representation of the procedure CFV

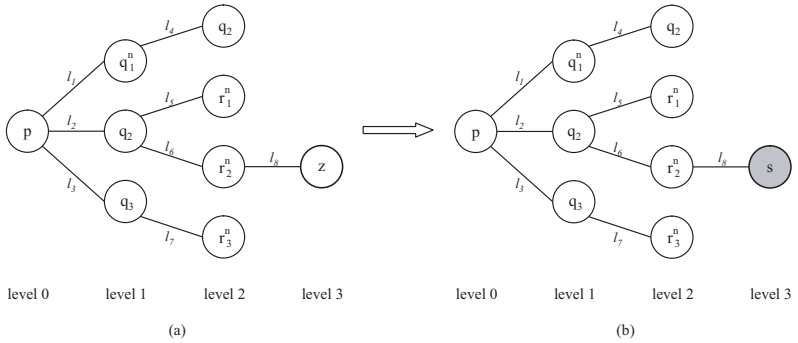


Figure 2. A sample tree of proving state predicate p with CFV

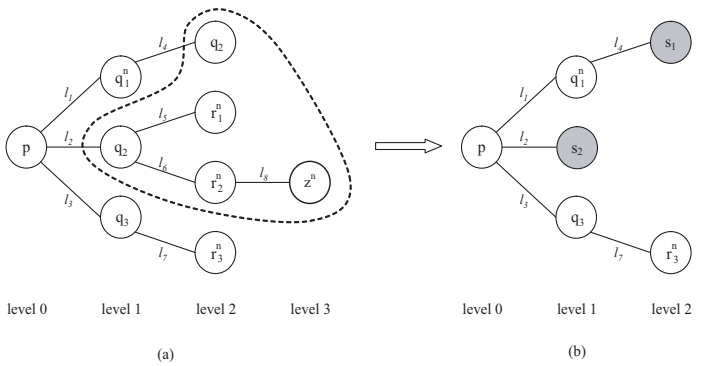


Figure 3. Another sample tree of proving state predicate p with CFV

Model Checking Executable Specification for Reactive Components

Bruno Blašković
Faculty of Electrical Engineering and Computing
Zagreb, Croatia
Email: bruno.blaskovic@fer.hr

***Abstract*—Finding design errors in the earliest phase of software developments is still challenging area of research. This paper deals with model checking of executable specification. Executable specification is introduced as *C* program. After that, *C* program is transformed into an input model for the Spin model checker. At the end, an example for the `Zune30` bug is presented.**

Keywords-executable specification; reactive component; software model checking; model transformation

I. INTRODUCTION

Telecommunication network software system can be modeled as the set of hierarchically connected communicating finite state automata (FSA). The basic unit of behavior is reactive component, modeled as FSA and referred as model $\mathcal{M}$ in this paper. FSA is implemented in C language subset. Such approach provides executable specification for component behavior analysis. In this paper, analysis is focused on component model checking. Executable specification can also serve as starting point for test cases definition, component simulator and target code skeleton generation. Component quality assurance is provided through *safety* ("Bad things will never happen") and *liveness* ("Good things will eventually happen") properties verification.

If property do not hold, component exhibits illegal behavior. Model checking approach define the model and check the properties of the model by means of assertions (invariants) and temporal logic formulas. In the case of illegal behavior, model checker provides counterexamples. Counterexample consists of a set of actions that describe paths (sequence of actions) to the errors.

FSA transitions describe dynamic behavior: C instructions are abstractions for internal actions like method calls or external actions like message sending/receiving events. There are no pointers or arrays in C code yielding straightforward translation; there is always the same FSA with different syntax representation.

First, designer defines FSA as C program. After that, C program is transformed to the form suitable for model checking. In short, FSA is designer's viewpoint about component behavior.

In order to model check or verify component behavior additional commands like *assertions* (`assert(<condition>)`) and *labels* are included into the program source. If all *assertions* are true, or if they are never false, program satisfies *"liveness"* property. The problem is where to put *assertions*: false *assertion* are hard to detect because that part of the code can be unreachable. Even the more, desired behavior can include another kind of *assertions* that can be true "many times", "infinitely often" or "only once". Such "*assertions*" are expressed with **L**inear **T**ime **L**ogic (LTL) formula. In Section VI-A, an example of linear time logic formula usage is presented.

Labels are used to check regular behavior and illegal behavior, respectively. Program is *"safe"* if "error" labels are always be unreachable, and "end" labels are eventually reachable. Program testing will not find all false *assertions*, so additional efforts are required. Model checker Spin has built–in facilities to detect assertion violations and unreached labels. In this paper we use model checker Spin to find unreached "end–state" and "non–progress" loops. Spin can also check concurrent errors from the specification. We expect separate study in order to extend specification with concurrent issue and to improve the abstraction for unbounded data values. Another set of problems are space–time limits. Space time limits are known as state explosion problem, because number of states grows exponentially. If only part of the system, consisting of several components is under consideration, state explosion problem can be under control.

This paper is organized as follows. After Introduction, in Section II related work is described. Scope and motivations are in Section III and theoretical background is presented in Section IV. After that, sketches of ψ–algorithms for model transformations are introduced in Section V. Description of a *C* source to the *Promela* code is in Section VI. The results of an experiment are given in Section VI-A, and, at the end, are conclusion and further research directions.

II. RELATED WORK

First, we introduce three approaches where specification in various formalism is translated into an model for model checkers. The origin of formal specification with Statechart is introduced by Harel in [1]. Statechart is an extension of FSA. Every statechart user defines specific properties that are hard to express in unique specification language, because statechart have no unique and clear semantic.

 ISBN: 978-1-61208-233-2

Object–oriented statechart semantic based on Statemate tool is formalized as labeled transition system in [2]. Graphic editing tool TCM can produce output to SMV, NuSMV and KRONOS model checkers. Bharadwaj and Heitmeyer [3] uses SCR (Software Cost Reduction) tabular notation as specification language. SCR specifications is transformed into an input model for Spin and SMV model checker. In all mentioned approaches [1] [2] [3], model definition phase prepares specification for model checkers. Our approach uses FSA encoded in C language. With such an approach, we avoid problems with semi–formal statechart semantic and the usage of specification languages outside UML set of diagrams.

Translating C programs to Promela [4] is another approach that checks C programs. A similar approach exists in [5] where Promela model is extended with direct inclusion of C code. Both approaches [4] and [5]) addresses C–code model checking. Our approach targets model checking of specification modeled as C program.

In [6] [7], `cbmc` C program model checker is described. C source is abstracted as Boolean program that is checked with satisfiability (SAT) tool. Our approach uses model extractor that is the part of `cbmc`. Extracted FSA follows BNF definition for FSA introduced later in Figure 4. We find this combination of tools useful because `cbmc` extracts models and Spin can check properties expressed in linear time temporal logic. At the end of this Section, the model checking fundamentals are introduced in [8]. Explicit state model checker Spin with industrial strength experience is described in [9]. Spin modeling language is called *Promela*.

III. Scope and motivation

It is well known fact that design errors like deadlock states, non–progress loops, illegal program termination, and message buffers overflow must be discovered as early as possible during the software life cycle. This paper is focused on executable specification analysis and the model transformation of executable specification to *Promela* model. For that purposes an illustrative example regarding `zune30 bug` has been selected from [10]. In our case, real scale example were components for e–Invoice service where an infinite loop has been discovered. `Zune30 bug` example has similar features as find in real scale examples, like unreached code or infinite loop. Similar piece of code with "small" programming mistake in telephone switch software canceled 50% of 133 million long distance calls.

The approach introduced within the paper bridges the gap between the tools capable of finding design errors and semi–formal specification. Usual approach for system specification is textual or semi–formal form, using UML or SDL+MSC diagrams. This paper starts from the C language model $\mathcal{M}$ as executable specification of state–transition system. It is designers responsibility to provide component model as much as possible close to the original. For that purpose, C language specification uses only small part of C language constructs that have direct implementation in Promela, because there is no need for pointers, complex data–type structures or arrays. After translation to Promela model, Spin [9] builds `pan` validator where checking procedures take place. Besides that, after model checker has proved desired properties, executable specification can be transformed to code skeleton (target language implementation).

Another possibility is to model specification as statechart and directly transform to the model that can understand the model checker. This approach yields several design inconsistencies:

– the semantic for a Statechart model of specification and the semantic of Promela model for the same specification is in general case different because specification can be interpreted in different ways,
– introducing executable specification as an intermediate representation (Figure 3.) provides the "simulator" for real application yielding information about overall system semantic and behavior, avoiding design inconsistencies,
– target code and model checking results are inconsistent without executable specification.

Instructions from the *C* language executable specifications are transitions that represents the real system behavior. Single transition is model or abstraction that describe method call, indivisible sequence of method calls, FSA execution or network of connected FSA executions. Although the model M, in most cases, describes single FSA, we can easily compose communicating FSA to the single higher level FSA using asynchronous product of FSA. Asynchronous product of FSA is built in feature of Spin (for details see [9] Appendix A). Each FSA is separate process in Promela model. Generic model $\mathcal{M}$ for reactive component or generic FSA or proces from Promela are syntactically different but semantically equivalent basic building block for component specification and definition. Transition $t_{\mathcal{M}}$ from Figure 3. describes the position of executable specification within the generic model, models behavior and unify transition semantic between all models. Each transition has the same form as Mealy FSA but with extended transition semantic (1).

$$\frac{input\_event}{output\_action} \tag{1}$$

Input_events are:

– guards, control–flow instructions, i.e., `if`
– message receiving events

Output_events are:

- message sending events,
- method calls,
- assignments,
- call of another FSA or FSA network.

 ISBN: 978-1-61208-233-2

IV. THEORETICAL BACKGROUND

Theoretical background is based on model $\mathcal{M}$ transformations [11] and consists of the following parts :

(1) "*Triptych*" environment for two–phase model transformations (Figure 2): (1) from high–level specification or requirements to executable specification and (2) from executable specification to verification (*Promela*–prml) or (2) to implementation code.
(2) model $\mathcal{M}$ for generic reactive component (Figure 3). Component is finite state automaton (FSA) with C language or Promela `proctype` construct representation,
(3) C program as executable specification (Section V-C). Executable specification can also be tested like any piece of C code,
(4) $\mathcal{M}_{tr}$ model transformation as framework for model checking executable specification (Figure 1),
(5) ψ algorithms for $\mathcal{M}_{tr}$ model transformations. Due to restricted instruction set in C specification, model transformations are simple `Perl` scripts.

We perform model checking for model $\mathcal{M}$ for property φ. Our approach follows usual approach [8] for model checking as described in Equation 2:

$$\mathcal{M}_{\mathtt{FSA}} \models \varphi_{\mathtt{LTL}} \tag{2}$$

A model $\mathcal{M}$ is an executable specification expressed as state–transition system or more precisely as extended FSA (eFSA). Extended FSA models:

– single eFSA,
– network of communicating FSA (cFSA),
– hierarchical network of communicating eFSA (hcFSA).

There is no universal approach for model checking executable specification. That means every domain is specific regarding designers or users requirements. As a consequence, we focus our attention to reactive software components generic model $\mathcal{M}$ (Figure 3) as the basic building block for FSA, extended FSA (eFSA), communicating FSA (cFSA) and hierarchical FSA (h(cFSA)).

From initial state s_0 (Figure 3) transition t_o initiates the component. There are two possible end states, regular (`end_OK`) and illegal (`end_NOK`), respectively.

Regular behavior is abstracted within the single transition $t_{\mathcal{M}}$. As previously said $t_{\mathcal{M}}$ can abstract the behavior of cFSA or h(cFSA). Illegal behavior is executed within $t_{\neg\mathcal{M}}$ transition. In regular cases FSA returns to initial state s_0 with

$$\mathcal{M}^{\alpha}_{\mathrm{spec}} \xrightarrow{\psi_1} \mathcal{M}^{\omega}_{\mathcal{C}} \xrightarrow{\psi_2:\mathrm{c2cfg;cfg2prml}} \mathcal{M}^{\pi}_{prml}$$

Figure 1. Model transformation sequence

t_{OK} transition while in illegal cases FSA returns to initial state with t_{NOK} transition (represented with dashed line on Figure 3), respectively. Following Equation 2. we introduce model $\mathcal{M}$ as triple in Floyd–Hoare logic and properties φ for *safety* and *liveness*:

$$\mathcal{M} \equiv \langle \{\mathtt{INV}\ pre\}\ \ code\ \ \{\mathtt{INV}\ post\}\rangle \longrightarrow \langle \varphi \equiv \Diamond\, \Box\, \mathtt{np\_}\rangle \tag{3}$$

Introduced linear time logicformula is checked with the `pan` analyzer of model checker Spin [9]. Executable specification

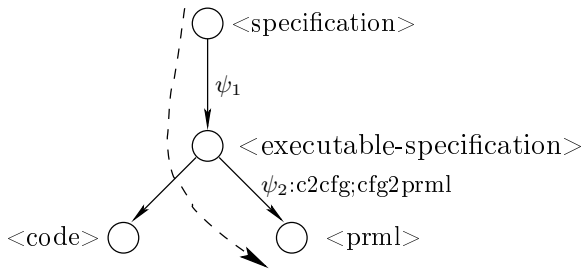


Figure 2. Triptych

is derived from top level semi–formal specification as described on Figure 2. (`<specification>` labeled circle). Top level specification (M_{spec} on Figure 1.) is transformed to executable specification with ψ_1 algorithm. In this paper

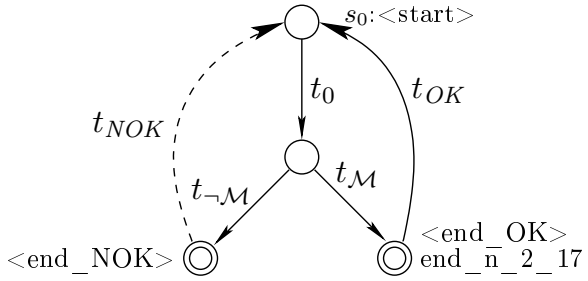


Figure 3. eFSA generic model $\mathcal{M}$ for Reactive Component

we focus our attention on translation of an executable specification to the *Promela* model. *Promela* model is the input to Spin model checker suitable for analysis of property φ from 3.

Code generation and transformation to executable specification are not the subject of this paper. The sequence of model transformation is summarized on Figure 1. In order to unify syntax representation for all internal FSA model transformation BNF representation is introduced in Figure 4.

V. ψ-ALGORITHMS

First, we introduce the definition of `ccfg`–`c` `control` `flow` `graph`. We shall refer `ccfg` simply as control flow graph `cfg`. A `cfg` is triple (S, T, L) where:

S set of states s_i, $s_i \in S$;
T (or $\longrightarrow$) is the set of transitions such that $T \subseteq S \times L \times S$
L is labeling functions (assign `C` instruction to the label l_j, $l \in L$

```
<eFSA> ::= <header> <body> | <comment>
 <header> ::= [h|H]  (.)* '\n'
 <comment> ::= # (.)* '\n'
 <body> ::= <keyw> <records>
  <records> ::= <record> '\n'
   <record> ::= <fields> <separator>
    <fields> ::= '[\w-_\(\)]'+
  <separator> ::= '\s'+ | '\t'+ | ':' | ','
  <keyw> ::= INIT | STATES | LABELS |
             TRANSITIONS} | FINAL | SL | LT
```

Figure 4. BNF for $\mathcal{M}$ FSA

Control flow graph `cfg` follows previously mentioned BNF syntax for `eFSA`, `cfg` derived from *C* source is presented in lisp–like form as the set of *state–label* pairs and the set of *state–arrow–next-state* triples, respectively:

$$(s_i, l_j)$$
$$(s_i \longrightarrow s_{i+i})$$

A. ψ_2–*c to cfg*

This algorithm (`c2cfg`) is model extraction [7] for `C` program. We use `goto-cc` model extractor introduced in [6].

B. ψ_2–*cfg to prml*

Control flow graph translation to *Promela* model (`cfg2prml`) algorithm consists of the following steps:

(1) substitute arrow $\rightarrow$ with label:
$(s_i \rightarrow s_{i+i}) \longrightarrow (s_i\, l_j\, s_{i+i})$
(2) abstract label l_j: $l_j \longrightarrow < l_j >$: **abstraction is already in C source.**
(3) $\forall s_i \in S$ substitute s_i with *Promela* `if` block or label abstraction
(4) "End of function" $\rightarrow$ `end_`

ψ_2–cfg to prml translation is realized as`Perl` script.

C. Example

As an example we present model $\mathcal{M}$ of `Zune30` bug. C program has been taken from [10] and translated to $\mathcal{M}_{prml}$ *Promela* model. This *C* program serves as executable specification model. Similar models $\mathcal{M}_{spec}$ of executable specification are derived from semi formal specification of distributed web applications, business processes and control software. For inputs like 366, 10593 `zune30.c` program enter endless loop. `Assertion` from line `15` with `Q1:` label is never executed in C program, yielding no *assert–violation*:

"$\mathcal{M}_C$ `C source for zune30`"

```
/* BUG: issue ./zune30 366, 10593 */
/* and have endless loop         */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int zune30(int days) {

  int year = 1980;
  while (days > 365) {
    if ((year % 4) == 0){
      if (days > 366) {
Q1:      assert(1);
        days = days - 366;
        year = year + 1;
      }
/*      else { */
/*      }      */
    }
    else {
      days = days - 365;
      year = year + 1;
    }
  }
  printf("%d\n", year);
  return 1;
}
```

After transformation *C* source ($\mathcal{M}_C$) is translated to *Promela* model ($\mathcal{M}_{prml}$). Analysis of *Promela* model $\mathcal{M}_{prml}$ gives the sequence of instructions that raise undesired behavior.

"$\mathcal{M}^{\pi}_{prml}$: `the Promela model`"

```
int UNKNOWN;
int year;
int days;
int cprntf;
int cgoto;
int creturn;
int cassertif;
int cassertFALSE;
int cassertTRUE;

active proctype acz() {

#if DAYS
   days=DAYS;
#endif

n_2_0: UNKNOWN=0; -> goto n_2_1;

n_2_1: year = 1980; ->  goto n_2_2;

n_2_2:
if
:: !(days > 365) -> goto n_2_15;   //true
::  (days > 365) -> goto n_2_3;   // false
fi;

n_2_15: cprntf=1 -> goto n_2_16;

n_2_3:
if
:: !(year % 4 == 0) -> goto n_2_12;  // true
::  (year % 4 == 0) -> goto n_2_4;   // false
```

```
fi;

n_2_16: creturn=2 -> goto end_n_2_17;
n_2_12: days = days - 365 -> goto n_2_13;

n_2_4:
if
:: !(days > 366) ->  goto n_2_11;  // true
::  (days > 366) -> goto n_2_5;    //  false
fi;

n_2_13: year = year + 1 -> goto n_2_14;
n_2_11: cgoto=3 -> goto n_2_14;

n_2_5:
if
:: cassertif=4 ->  goto n_2_8;   //true
:: cassertif=5 ->  goto n_2_6;   //false
fi;

n_2_14: cgoto=6 -> goto n_2_2;
n_2_8: cassertFALSE=7 -> goto n_2_9;
n_2_6: cassertTRUE=8 -> goto n_2_7;
n_2_9: days = days - 366 -> goto n_2_10;
n_2_7: cgoto=9 -> goto n_2_9;
n_2_10: year = year + 1 -> goto n_2_11;

end_n_2_17: skip;    // End of Function
}
```

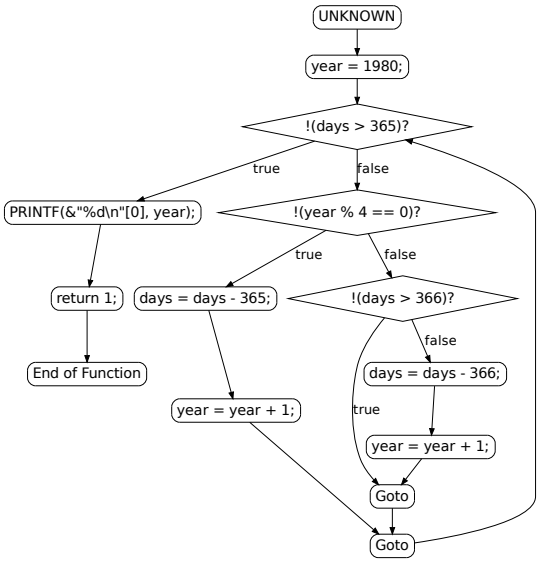


Figure 5. C Control flow graph cfg for zune30 example

Next section will explain transformation from *C* source to *Promela* model.

VI. Model transformation: from *C* to *Promela*

Model transformation is performed following the theoretical concepts from the Section IV and Figure 1. The first step is call to goto-cc that implements transformation of *C* source to control flow graph cfg. ($\mathcal{M}^{\omega}_{cfg} \longrightarrow \mathcal{M}^{\omega}_{cfg}$). Transformation is implemented in ψ:c2cfg algorithm.

Vertexes from Figure 5 are executable instructions and edges are "connections" between instructions, respectively. Nodes are assignments like year=year+1 or if statements (for example: if(days >365). In real situations additional assignments are method calls. We assume that methods are safe and live, thus always return desired values. That means methods have *assume-guarantee* property that is checked separately.

The result of the transformation is coded in lisp–like syntax:

```
_ "M^ω_cfg cfg for zune30 in lisp--like syntax" _

SL
(n_2_0  UNKNOWN)
(n_2_1  "year = 1980;")
(n_2_2  "!(days > 365)?")
(n_2_15  "PRINTF("%d\n",year)")
(n_2_3  "!(year % 4 == 0)?")
(n_2_16  "return 1;")
(n_2_12  "days = days - 365;")
(n_2_4  "!(days > 366)?")
(n_2_17  "End of Function")
(n_2_13  "year = year + 1;")
(n_2_11  Goto)
(n_2_5  "!(_Bool)1?")
(n_2_14  Goto)
(n_2_8  "Assert(FALSE)")
(n_2_6  "(void)0;")
(n_2_9  "days = days - 366;")
(n_2_7  Goto)
(n_2_10  "year = year + 1;")

LT
(n_2_0 -> n_2_1)
(n_2_1 -> n_2_2)
(n_2_2 -> n_2_15  true)
(n_2_2 -> n_2_3   false)
(n_2_15 -> n_2_16)
(n_2_3 -> n_2_12 true)
(n_2_3 -> n_2_4  false)
(n_2_16 -> n_2_17)
(n_2_12 -> n_2_13)
(n_2_4 -> n_2_11 true)
(n_2_4 -> n_2_5  false)
(n_2_13 -> n_2_14)
(n_2_11 -> n_2_14)
(n_2_5 -> n_2_8 true)
(n_2_5 -> n_2_6 false)
(n_2_14 -> n_2_2)
(n_2_8 -> n_2_9)
(n_2_6 -> n_2_7)
(n_2_9 -> n_2_10)
(n_2_7 -> n_2_9)
(n_2_10 -> n_2_11)
```

For example, vertexes n_2_1 is assignment for *C* statement year=1980 and transitions between vertexes are triples (n_2_1 $\longrightarrow$ n_2_2), respectively.

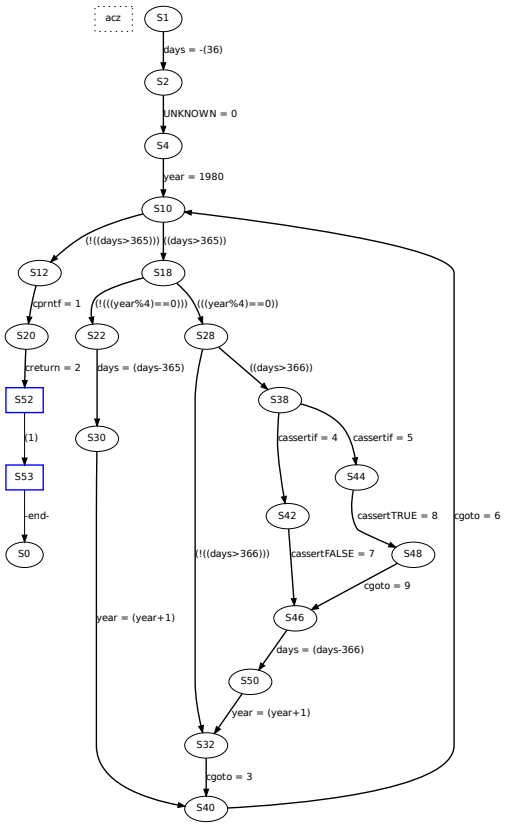


Figure 6. FSA for Promela model $\mathcal{M}^{\pi}_{prml}$

After that, algorithm ψ_2:*cfg2prml* is applied, resulting in *Promela* model $\mathcal{M}^{\pi}_{prml}$ as presented in Section. V-C. The algorithm translates control flow graph to *Promela* code $\mathcal{M}^{\pi}_{cfg} \longrightarrow \mathcal{M}^{\pi}_{prml}$. The *Promela* model is another invariant form of $\mathcal{M}^{\pi}_{C}$ model, Figure 6 visualize it as an finite state automaton. Spin's verifier `pan` has options that enable visualization of *Promela* models as automaton.

There are significant difference from `cfa` from figure 5, instructions l_j are placed on transition labels and many instructions have abstracted form $< l_j >$, for example, `assert` is replaced with `cassertif=4` abstracted form.

Next step is *Promela* model analysis of liveness and safety properties.

A. Experiment result analysis

The analysis of *Promela* model from Section V-C yields the following results:

– there are unreached portions of code
– there are endless loops

In order to achieve this results two verifier runs are required:

– Non–progress cycles (loops) are detected with linear temporal logic formula: $\Diamond\Box$`np_`, where `np_` is *Promela* built–in variable for marking the progress of global system state status. In our example, formula is *false* producing counterexample with non–progress cycle.
– unreached instruction from *Promela* model ("dead–code") is standard built–in function into the *pan* verifier.

The output from the *pan* verifier is counterexample with the path to the error. Each row presents the line number of instruction from the *Promela* model presented in Section V-C. Non–progress loop is sequence of instructions with line numbers `24 32 40 45 53 24 32 40 ...`

```
_____________ "non-progres loops" _____________
1
2   z30.ltg.prml:14          [days = 366]
3                 days = 366
4   z30.ltg.prml:17          [UNKNOWN = 0]
5   z30.ltg.prml:19         [year = 1980]
6         year = 1980
7   <<<<<START OF CYCLE>>>>>
8   z30.ltg.prml:24          [((days>365))]
9   z30.ltg.prml:32   [(((year%4)==0))]
10  z30.ltg.prml:40          [(!((days>366)))]
11  z30.ltg.prml:45          [cgoto = 3]
12        cgoto = 3
13  z30.ltg.prml:53         [cgoto = 6]
14        cgoto = 6
15 spin: trail ends after 16 steps
16                year = 1980
17                days = 366
```

Counterexample pointing unreached code use `pan` verifier built in options for unreached code detection. Each row presents the line number of unreached instruction from *Promela* model presented in Section V-C (27, 35, 36, 44, ...).

```
____________ "unreached end--state" ____________
1
2 unreached in proctype z30
3
4   z30.ltg.prml:27,  "cprntf = 1"
5   z30.ltg.prml:35,  "creturn = 2"
6   z30.ltg.prml:36,  "days = (days-365)"
7   z30.ltg.prml:44,  "year = (year+1)"
8   z30.ltg.prml:44,  "year = (year+1)"
9   z30.ltg.prml:49,  "cassertif = 4"
10  z30.ltg.prml:49,  "cassertif = 5"
11  z30.ltg.prml:54,  "cassertFALSE = 7"
12  z30.ltg.prml:55,  "cassertTRUE = 8"
13  z30.ltg.prml:56,  "days = (days-366)"
14  z30.ltg.prml:57,  "cgoto = 9"
15  z30.ltg.prml:58,  "year = (year+1)"
16  z30.ltg.prml:61,  "-end-"
17        (11 of 53 states)
```

VII. Conclusion and Further Work

We have presented model checking of specification as software model checking for C language.

We find that Spin model checker is feasible solution because Spin finds deadlocks, unreached code, assertion violations, invalid end states, and analyze linear time logicformula. In the same time, executable specification can be

analyzed, tested as every C program. Our approach avoids complex and long term development of model extractor with tools like CIL. Another benefit is the application of linear time logicformula on C specification. Usual approach puts assertions in the code in the place according to the designer's discretion. Sometimes it is necessary that assertion is true to "some point in the future infinitely often" which can be expressed as temporal logic formula. With our approach linear time logic formula is the part of Promela model and consequently also the part of C specification. State explosion and designer mistakes during specification definition are still problems that needs improvements. "Designers will never use it" syndrome is always the problem when introducing development paradigms.

Further work will focus on more rigid data–types consistency check. That requires formal development of abstract data structures. In most cases, such data structures are defined over infinite domains so further refinements should avoid infinite data domains, or introduce data abstractions.

Besides Spin, the comparison with other model checkers, like Petri net tools, could improve verification. Model checkers search for solutions within finite space, the improvement of model checking with unbounded parameters (`days` in our example) yields: $\mathcal{M}(days) \models \varphi$.

Bounded model checking [12] and the usage satisfiability modulo theory (SAT [13] and SMT [14]) solvers are the promising research direction.

Automated code generation from executable specification is another possible direction for research. The most promising is TDD "Test Driven Development" because code skeleton is populated with test case commands.

REFERENCES

[1] D. Harel, "Statecharts in the making: a personal account," in *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*. San Diego, California: ACM, 9-10 June 2007, pp. 1–43.

[2] R. Eshuis, D. N. Jansen, and R. Wieringa, "Requirements-level semantics and model checking of object-oriented statecharts." *Requirements Engineering*, vol. 7, no. 4, pp. 243–263, 2002.

[3] R. Bharadwaj and C. L. Heitmeyer, "Model Checking Complete Requirements Specifications Using Abstraction," *Automated Softwware Engineering*, vol. 6, no. 1, pp. 37–68, 1999.

[4] K. Jiang, "Model Checking C Programs by Translating C to Promela," Master's thesis, Uppsala Universitet, Department of Information Technology, 2009.

[5] G. J. Holzmann, "Logic Verification of ANSI-C Code with SPIN," in *SPIN Model Checking and Software Verification*, ser. Lecture Notes in Computer Science, K. Havelund, J. Penix, and W. Visser, Eds., vol. 1885, 7^{th} International SPIN Workshop. Stanford CA USA: Springer, August 2000, pp. 131–147.

[6] "CBMC is a Bounded Model Checker for ANSI-C," (last time visited July, 5^{th}2012). [Online]. Available: http://www.cprover.org/cbmc

[7] 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.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, January 1999.

[9] G. Holzmann, *Spin model checker, the: primer and reference manual*, 1st ed. Addison-Wesley Professional, 2004.

[10] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[11] A. Metzger, "A systematic look at model transformations," in *Model-Driven Software Development*, S. Beydeda, M. Book, and V. Gruhn, Eds. Springer Berlin Heidelberg, 2005, pp. 19–33.

[12] Armin Biere and Alessandro Cimatti and Edmund M. Clarke and Ofer Strichman and Yunshan Zhu, "Bounded Model Checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[13] Biere, Armin and Heule, Marijn J. H. and van Maaren, Hans and Walsh, Toby, Ed., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.

[14] Armando, Alessandro and Mantovani, Jacopo and Platania, Lorenzo, "Bounded model checking of software using SMT solvers instead of SAT solvers," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 69–83, Jan. 2009.

 ISBN: 978-1-61208-233-2

Software Architectural Drivers for Cloud Testing

Etiene Lamas, Luiz Alberto Vieira Dias, Adilson Marques da Cunha
Computer Science Division
Aeronautics Institute of Technology, ITA
Sao Jose dos Campos, Brazil
{etiene, vdias, cunha}@ita.br

Abstract—**This paper focuses on the research issues that Cloud Computing imposes on Software Testing. For this purpose, Cloud Testing can be defined as a Software Testing method based on Cloud Computing technology. Software Testing has been an important component within the development process. In order to face the rapid growth of Cloud Computing, Reference Architectures provide a simple and organized environment for applications development. The outage on Cloud Services must be considered an exception not a rule. This research emphasizes the complexity of Cloud Testing, in order to prevent services disruption, as it happened, for example, with the Amazon in April 2011. This research aims to investigate, design, implement, and propose key Software Architectural Drivers for Cloud Testing, focusing on monitoring the quality. Cloud Testing integration may allow monitoring products and services with efficient deliverables. The main advantage that arises from these proposed drivers is the provision of Cloud Testing Reference Architectures to be applied in practice. The main contribution of software architectural drivers is the quantitative monitoring of quality for both end products and services.**

Keywords-cloud testing; software architectural drivers; testing of cloud services; testing of cloud products; reference architectures

I. INTRODUCTION

In general, Cloud Computing changes the way Information Technology (IT) services are delivered. To monitor these changes, Cloud Testing can be defined as a Software Testing method based on Cloud Computing technology [1].

Parveen and Tilley [2] show that not all applications are suitable for testing in the Cloud and nor all types of testing are suitable for the Cloud.

The outage on Cloud Services must be considered an exception not a rule. This research emphasizes the complexity of Cloud Testing, in order to prevent services disruption, as it happened, for example, with the Amazon in April 2011 [3]. This research aims to investigate, design, implement, and propose key Software Architectural Drivers for Cloud Testing (SADCT), focusing on monitoring quality. Thus, an investigation about the generic and the specific theory has been conducted.

The drivers proposed by the authors for Reference Architectures (RAs) are set in order to quantitative monitor Quality of Products (QoP) and Quality of Services (QoS) in the Cloud. The main advantage arising from these proposed drivers is to provide Cloud Testing Reference Architectures to be applied in practice. The main problem is how to monitor and evaluate quantitatively the quality of the Cloud Testing. The main contribution of software architectural drivers is the quantitative monitoring of quality for both end products and services.

This article is organized as follows. Section 2 introduces Reference Architectures. Section 3 describes basic Cloud Computing concepts. Section 4 emphasizes the importance of Cloud Testing and presents the testing of Cloud Services. Section 5 specifies the Cloud Testing Reference Architectures. Section 6 proposes its key architectural drivers. Section 7 includes a Proof of Concept (PoC) study. Finally, Section 8 highlights some conclusions and future works.

II. REFERENCE ARCHITECTURES

In order to face the rapid growth of Cloud Computing, Reference Architectures provide a simple and organized environment for applications development.

A Reference Architecture (RA) is the generalized architecture of several end systems that share one or more common domains. The Reference Architecture defines the common infrastructure to the end systems and also the interfaces of components that will be included in the end systems. The Reference Architecture is then instantiated to create software architecture of a specific system [4].

The principles governing the design and evolution of a system and also the relationships between their components and the environment can be found in a Reference Architecture, which represents its fundamental organization [5].

To facilitate the understanding of the operational intricacies in Cloud Computing, the overview of its Reference Architecture will be presented in the following section.

III. BASIC CLOUD COMPUTING CONCEPTS

Given the rapid growth in its use, it is necessary to define Cloud Computing and Cloud Computing Reference Architectures.

 ISBN: 978-1-61208-233-2

A. Cloud Computing Definition

According to the National Institute of Standards and Technology (NIST) [6], Cloud Computing consists of service models, deployment models, and essential characteristics.

This definition is widely accepted as a valuable contribution toward providing a clear understanding of Cloud Computing technologies and Cloud Services.

It provides a simple and unambiguous taxonomy of three service models available to Cloud Consumers: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

It summarizes the four deployment models describing how the computing infrastructure that delivers these services can be shared: Private Cloud, Community Cloud, Public Cloud, and Hybrid Cloud.

Finally, the NIST definition also provides a unifying view of five essential characteristics that all Cloud Services exhibit: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service [6].

The three service models identified by the NIST, i.e., SaaS, PaaS, and IaaS, offer to the consumers different types of service management operations and expose different entry points into Cloud Systems.

B. Cloud Computing Reference Architecture

The overview of the NIST Cloud Computing Reference Architecture [7] is a logical extension to the NIST Definition of Cloud Computing.

According to Liu et al. [7], it is a generic high-level conceptual model that is an effective tool for discussing the requirements, structures, and operations of Cloud Computing. Also, according to [7], it defines a set of actors, activities, and functions that can be used in the process of developing cloud computing architectures. It describes five major actors with their roles and responsibilities, using the newly developed Cloud Computing Taxonomy. The five major participating actors are: (i) Cloud Consumer - a person or organization that maintains a business relationship with, and uses service from, Cloud Providers; (ii) Cloud Provider - a person, organization, or entity responsible for making a service available to interested parties; (iii) Cloud Broker - an entity that manages Cloud Services; (iv) Cloud Auditor - a party that can conduct independent assessment of Cloud Services; and (v) Cloud Carrier - an intermediary that provides connectivity and transport of Cloud Services. Each actor is an entity (a person or an organization) that participates in a transaction or process and/or performs tasks in Cloud Computing [7].

The NIST Cloud Computing Reference Architecture [7] focuses on the requirements of "what" Cloud Services provide, not on "how to" design solutions and implementations.

In order to improve the quality of Cloud Services, the interactions between the actors in Cloud Testing scenarios will be discussed in the following section.

IV. TESTING OF CLOUD SERVICES

Software Testing has been an important component within the development process. This paper focuses on the research issues that Cloud Computing imposes on Software Testing.

The architecture described by Blokland and Mengerink [8] consists of detailed risks that may occur when one starts using Cloud Computing, grouped into themes. Next to these risks, in their book, there are sets of test measures. Some measures do exist, like load testing, but polished to fit the new needs for applying performance testing in the Cloud.

The important asset of [8] is the link made from each individual risk to the different measures needed to cover the risk.

According to Blokland and Mengerink [8], there are also new measures that stretch the definition of test, like test in production. These measures are new because they are Cloud specific and present the complexity of testing in the Cloud.

It should be emphasized that some aspects of quality can be tested "on live" and it is very wise to continuously test them, because of the ever-changing situation in the Cloud Environment.

Testing activities must continue even after the system has gone live. But, there are other aspects that should be 'more traditionally' tested, before a new version of the system is put into the live Cloud Environment. Testing is not done only under the main implementation phases (Unit Testing; Integration Testing; System Testing; and Acceptance Testing) as it used to be, but it will be done also during selection (when the Cloud Services are selected). The criteria for selection are chosen for mitigating risks.

Because once in Cloud production everything might change, it is needed to continuously test the software under production. Some measures are specific for the Cloud, like how to deal with rules and regulations in different countries [7], like Migration Testing.

When software is running "in house", most of the failures are under control; but when "in the Cloud" everything is different, because failures are not under control any more. Due to the mutability of the Cloud Environment, it is necessary to verify if the services are still working after the deployment. The testing under production will validate the functionalities in this environment.

V. CLOUD TESTING REFERENCE ARCHITECTURE

Architectural Drivers are defined as the major quality attribute goals that shape the Cloud Reference Architectures [9]. This research aims to investigate, design, implement,

 ISBN: 978-1-61208-233-2

and propose key Software Architectural Drivers for Cloud Testing, focusing on monitoring quality.

Aiming to understand Reference Architecture roles for Cloud Testing, Figure 1, adapted from [10], presents the interaction of the software tester role (Cloud Tester) with the Cloud Environment specified for this research. Figure 1 also highlights those that provide and consume services.

Notice that IaaS supports PaaS that, in turn, supports SaaS.

The main characteristics that distinguish Cloud Testing from regular Software Testing are related to risks across all three layers of the Cloud stack (IaaS, PaaS, and SaaS), as seen in Figure 1. It is important to keep this basic stack in mind as the building blocks of the Cloud Computing system.

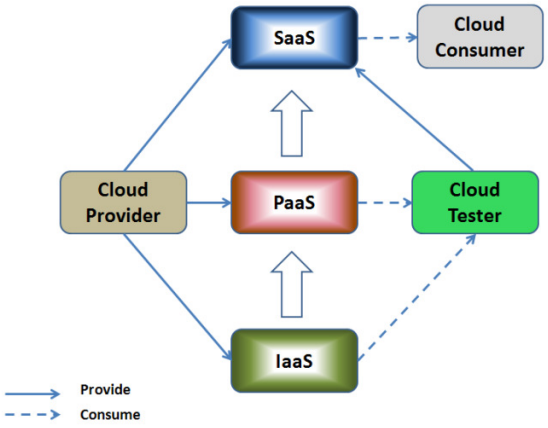


Figure 1. The Reference Architecture roles for Cloud Testing.

The Cloud Provider is responsible for providing, managing, and monitoring the entire structure to the solution of Cloud Computing, freeing the Cloud Tester and the Cloud Consumer from these types of liability. To do this, the Cloud Provider provides services for Cloud Consumers.

According to Veras [10], this organization in roles helps to define the actor and their different interests on Cloud Computing. Actors can assume different roles at the same time, according to their interests, and only the Cloud Provider supports all three functions of Cloud Services (IaaS, PaaS, and SaaS). From the viewpoint of interaction, among the three functions of service, IaaS provides computing resources (hardware or software) to PaaS. In its turn, PaaS provides resources, technologies, and tools for the development and the delivery of services to be implemented, becoming available as SaaS.

It is important to mention that an organization that provides Cloud Services needs does not necessarily provide all three-service functions. That is, a Cloud Provider can provide the option IaaS without necessarily also providing a PaaS [10].

This actor/role-based model is intended to serve the expectations of the stakeholders by allowing them to understand the overall view of roles and responsibilities, in order to assess and assign risks [7].

VI. THE KEY ARCHITECTURAL DRIVERS

According to Kazman et al. [9], the project manager describes what business goals are motivating the development effort and hence what will be the primary Architectural Drivers (e.g., high availability, time to market, or high security).

Aiming to understand the key architectural drivers for Cloud Testing Reference Architecture, the purpose of Figure 2, suggested by the authors, is to provide the guidance for the Cloud Testers to acquire knowledge on all needed testing categories.

Some proposed drivers for Cloud Testing Reference Architectures were based on traditional concepts that allow products with recognized quality (QoP), and another proposed drivers, specific for the Cloud, that allow better quality for Cloud Services (QoS).

These drivers were defined based on concepts from the traditional testing management environment, as seen in the bottom side of Figure 2, and also on concepts and elements, for the Cloud Testing management environment, as seen in the upper side of Figure 2.

A. *Traditional testing management environment*

The most important concepts for traditional test management environment can be clustered in two groups. The first group of concepts involves a set of definitions to support the Cloud Testing with Noncloud standards. These definitions relate to the guidelines for Software product Quality Requirements and Evaluation (SQuaRE) [11] and the appropriate breadth and depth of test documentation. The second group of concepts involves a set of methods supporting Cloud Testing with Noncloud methodologies. These definitions relate to effective methods for Software Testing [12] and these specific methods are listed below.

The authors suggest the use of an Agile Software Development Methodology, in order to deliver as much quality software as possible, within a series of short time boxes called Sprints, which last about a month. This methodology is characterized by short, intensive, and daily meetings involving the whole developers' team [13]. Agile is iterative and incremental. This means that the testers must test each increment of coding as soon as it is finished [14].

Finally, the second group of concepts involves also a set of techniques supporting Cloud Testing with Noncloud techniques. These are related to functional and structural techniques for Software Testing. These groups are:

a) Noncloud Standards: In this group of drivers, as seen in the bottom left side of Figure 2, the standards ISO/IEC 25000 named Software product Quality

 ISBN: 978-1-61208-233-2

Requirements and Evaluation (SQuaRE) should be applied [11], and the IEEE Std 829-2008, named IEEE Standard for Software and System Test Documentation, should be also applied [15]; and

b) Noncloud Testing Methodologies and Techniques: In this group of drivers, in the bottom right side of Figure 2, traditional phased software agile methodologies and effective methods for Software Testing should be applied. Traditionally, most of the test effort occurs after the requirements have been defined and the coding process has been completed. But, in the Agile approaches [14], most of the test effort is on-going. Newer development models, such as Agile, often employ Test Driven Development (TDD) and place an increased portion of the testing in the hands of the developer, before it reaches a formal team of testers. In a more traditional model, most of the test execution occurs after the requirements have been defined and the coding process has been initiated [14]. Also, in this group of drivers, in the bottom right side of Figure 2, the techniques can be divided into functional and structural. The main functional system testing techniques are: (i) Requirements - system performs as specified; (ii) Regression - verifies that anything unchanged still performs correctly; (iii) Defects Handling - defects can be prevented or detected, and then corrected; (iv) Manual support - the people-computer interaction works; (v) Control - controls reduce system risk to an acceptable level; and (vi) Parallel - old system and new system run and their results are compared to detect unplanned differences [12]. The main structural testing techniques are: (i) Stress - system performs with expected volumes; (ii) Execution - system achieves desired level of proficiency; (iii) Recovery - system can be returned to an operational status after a failure; (iv) Operations - system can be executed in a normal operational status; (v) Compliance - system is developed in accordance with standards and procedures; and (vi) Security - system is protected in accordance with the importance to organization [16].

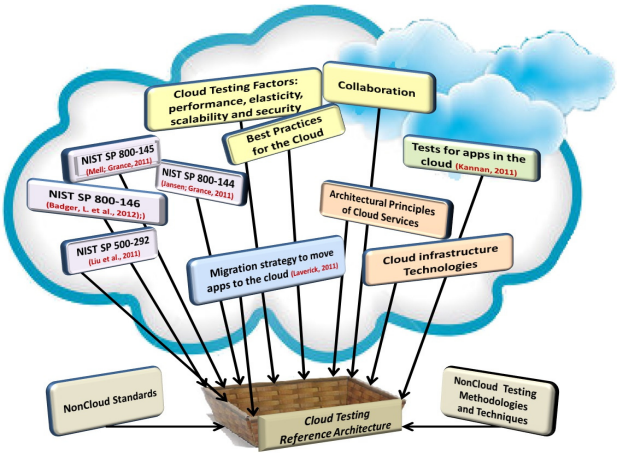


Figure 2. Software Architectural Drivers for Cloud Testing (SADCT).

B. Cloud Testing management environment

The most important concepts and elements for the Cloud Testing management environment can be clustered in five groups. The first group is a set of definitions to support the Cloud Testing with standards. These are related to adoption, development, and provision of testing and security for Cloud Computing. The second group is the set of best practices supporting Cloud Testing with collaboration and relevant factors. These are related to the Cloud Environment and essential characteristics for Cloud Services. The third group is a set of techniques supporting Cloud Testing with challenges. These are related to the testing techniques. The fourth group is a set of concepts supporting Cloud Testing with architectural principles. These are related to the technologies comprised in the Cloud Infrastructure. In the Cloud, not all applications are equally created. Finally, the fifth group is a set of steps supporting Cloud Testing with strategy for porting applications to the Cloud. These steps are related to Cloud Migration strategies. These groups are:

a) Cloud Standards: In the grey group of drivers, in Figure 2, the NIST Definition of Cloud Computing should be applied [6]; the NIST Guidelines on Security and Privacy in Public Cloud Computing should be applied [16]; the NIST Cloud Computing Synopsis and Recommendations should be applied [17]; and the NIST Cloud Computing Reference Architecture should be also applied [7]. The logical step to take after the formation of the NIST Cloud Computing definition is to create an intermediate reference point from where one can frame the rest of the discussion about Cloud Computing and begin to identify sections in the Reference Architecture in which standards are either required, useful, or optional [7];

b) Testing Factors, Collaboration, and Best Practices for the Cloud: In the yellow group of drivers, in Figure 2, it is important to mention that testing for Cloud-based applications presents its own specific challenges. Understanding how these applications are structured goes a long way in designing and executing appropriate test plans for them. These tests are done in addition to the usual Unit; Integration; System; Acceptance, and Performance. For example, the Performance should be achieved in the Cloud by testing for bandwidth, connectivity, scalability, and quality of the end-user experience. When testing Cloud applications, it is needed to validate and verify specific Cloud functionalities such as redundancy, failover, and performance scalability. Also, in the yellow group of drivers, suggested by the authors, it is important to mention that the Cloud provides an environment that supports global collaboration and knowledge sharing, as well as, group decision-making. Shared sites can be easily set up, replicated, and torn down as needed to meet the collaboration requirements of a given project. For collaboration, the best practices are to: (i) continuously monitor from users' perspective and end-user response time; (ii) implement end-to-end diagnostics; (iii) design for fault-tolerance; and (iv) load test to determine the breaking point.

 ISBN: 978-1-61208-233-2

For performance, the best practices are to: (i) understand where all bottlenecks are; (ii) mitigate bottlenecks; (iii) test performance for understanding normal and peak load to baseline "normal"; and (iv) continuously monitor performance from users' perspective. For scalability, the best practices are to: (i) architect for elasticity; (ii) use an elastic platform to scale services and data; (iii) isolate functions to scale them separately; (iv) implement a Cloud bursting strategy for load balancing between Clouds [17]; (v) automate scaling to quickly scale-up and down; and (vi) execute the load test in your application;

c) Testing techniques: In the green group of drivers, in Figure 2, Kannan [18] exemplifies challenges for: (i) browsers testing; (ii) service provisioning/de-provisioning testing; (iii) distributed Cloud Testing; (iv) multi-tenancy testing; (v) Cloud Portability Testing; among others. Cloud-based software applications have some additional characteristics compared to Noncloud-based ones. These pose additional challenges but with a systematic and comprehensive approach to test planning, to be appropriately handled;

d) Cloud infrastructure and architectural principles: In the pink group of drivers, in Figure 2, also suggested by the authors, it is important to mention that Cloud infrastructure should never go down for a day. Clouds are characterized by various technologies including: (i) virtualization (hypervisor); (ii) automation; (iii) monitoring; and (iv) service portal/service catalog. Currently, there are Cloud architectural principles for high availability: (i) monitoring; (ii) fault tolerance; and (iii) fixable.

e) Migration strategies: In the blue group of drivers, in Figure 2, it is important to mention that in the Cloud, not all applications are created equal, and some are completely wrong for the infrastructure model. To make the right decision about which applications to move, it is needed a solid migration strategy. It is also needed to consider the application portfolio and the business requirements to prevent problems such as poor application performance and latency, data leakage, or issues with compliance or other regulations [19]. Here is how to develop a foolproof strategy for moving the right applications to the cloud, which starts by outlining clear objectives, then focuses on your application portfolio's characteristics and business requirements to determine the best fit. These steps ensure that moving to the Cloud will be possible.

VII. PROOF OF CONCEPT (POC)

A Proof of Concept (PoC) is an exercise to test a design idea or assumption. Software developers tend to utilize PoCs instinctively when they experiment with technology.

The presented drivers could be used in a PoC to quantify the quality monitoring throughout the key Software Architectural Drivers for Cloud Testing (SADCT). This will be elaborated using the Multi-Attribute Global Inference of Quality (MAGIQ) technique for Software Testing [20]. The MAGIQ technique uses Rank Order Centroids (ROC) [21] to convert system comparison drivers into normalized numeric weights, and then computes an overall measure of quality as a weighted (by comparison drivers) sum of system ratings.

The PoC was applied to an academic project named "Fraud Detection and Unauthorized Access (FDUA)", developed at the Brazilian Aeronautics Institute of Technology (*Instituto Tecnologico de Aeronautica* - ITA) aiming to evaluate the feasibility of the Software Architectural Drivers for Cloud Testing propositions.

Given the FDUA Test Scenario and the Software Architectural Drivers for Cloud Testing Hierarchical Diagram, as seen in Figure 3, suggested by the authors, the students (Cloud Testers), all seasoned testers, were asked to rank the Software Architectural Drivers for Cloud Testing items.

At this point, the Software Architectural Drivers for Cloud Testing Hierarchical Diagram was performed as a hierarchical decomposition of the proposed Software Architectural Drivers for Cloud Testing by using MAGIQ technique for Software Testing [20].

In the MAGIQ analysis technique, after the attributes of the systems under evaluation have been determined, rank order centroids are used to assign relative weights to each comparison attribute [20].

The Cloud Testers examine the comparison attribute set at each level of the hierarchical decomposition of the attributes, and ranks the Software Architectural Drivers for Cloud Testing in the set from most important to least important, and then assigns relative weights to each Software Architectural Drivers for Cloud Testing using rank order centroids [21].

For each item, the Cloud Testers should assign a weight (in the range 0 to 1), which will be composed with the MAGIQ coefficients, in order to evaluate quantitatively QoP and QoS.

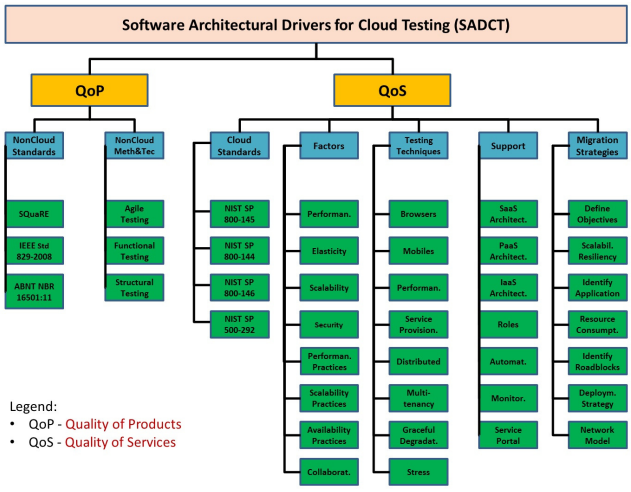


Figure 3. The SADCT Hierarchical Diagram.

Numerically, the quality of Cloud Testing is obtained by Software Architectural Drivers for Cloud Testing.

Equation (1), proposed by authors, quantitatively monitors the quality of Cloud Testing:

$$SADCT = QoP + QoS. \quad (1)$$

Software Architectural Drivers for Cloud Testing (SADCT) = 0.910																				
Quality of Products (QoP) (Rank = 0.750) (Result = 0.719)						Quality of Services (QoS) (Rank = 0.250) (Result = 0.191)														
Ranks	Weight	Part.Res.	Ranks	Weight	Part.Res.	Ranks	Weight	Part.Res.	Ranks	Weight	Part.Res.	Ranks	Weight	Part.Res.	Ranks	Weight	Part.Res.	Ranks	Weight	Part.Res.
0.250	100%	0.25	0.750	83%	0.708	0.124	88%	0.118	0.457	75%	0.435	0.257	47%	0.097	0.124	57%	0.107	0.040	20%	0.008
0.111	1	0.028	0.611	1	0.458	0.104	0.5	0.006	0.184	1	0.084	0.016	0	0.000	0.192	1	0.024	0.299	0.2	0.002
0.611	1	0.153	0.111	0.5	0.042	0.521	1	0.065	0.025	0	0.000	0.079	0.25	0.002	0.192	1	0.024	0.156	0.2	0.001
0.278	1	0.070	0.278	1	0.209	0.104	1	0.013	0.184	1	0.084	0.236	1	0.029	0.109	1	0.014	0.299	0.2	0.002
						0.271	1	0.034	0.340	1	0.155	0.236	1	0.029	0.370	1	0.046	0.059	0.2	0.000
									0.081	1	0.037	0.111	0.5	0.007	0.032	0	0.000	0.109	0.2	0.001
									0.081	1	0.037	0.054	0	0.000	0.032	0	0.000	0.020	0.2	0.000
									0.081	1	0.037	0.033	0	0.000	0.073	0	0.000	0.059	0.2	0.000
									0.025	0	0.000	0.236	1	0.029						
Noncloud Standards			Noncloud Meth&Tech			Cloud Standards			Factors			Cloud Testing Techniques			Support			Migration Strategies		

Figure 4. The Q1 Results.

The drivers for traditional concepts will focus on Quality of Products (QoP), and the additional drivers, specific for the Cloud, will focus on the Quality of Services (QoS).

VIII. PoC Results

The Proof of Concept (PoC) was applied in four different Agile Testing Quadrants or phases (Q1, Q2, Q3, and Q4) of the Cloud Testing [14].

Figure 4, suggested by the authors, is a data sheet in order to calculate values for Quality of Products (QoP) and Quality of Services (QoS), applied to the Software Architectural Drivers for Cloud Testing, as in (1):

a) Q1 – In the Unit Testing, the results are obtained through the calculations from the data sheet presented in Figure 4. Summarizing, the value for the obtained Software Architectural Drivers for Cloud Testing is 0.910, because QoP is 0.719 and QoS is 0.191;

b) Q2 – In the Integration Testing, the results are obtained through similar calculations. Summarizing, the value for the obtained Software Architectural Drivers for Cloud Testing is 0.715, because QoP is 0.563 and QoS is 0.151;

c) Q3 – In the System Testing, the results are obtained through similar calculations. Summarizing, the value for the obtained Software Architectural Drivers for Cloud Testing is 0.830, because QoP is 0.682 and QoS is 0.149; and

d) Q4 – In the Acceptance Testing, the results are obtained through similar calculations. Summarizing the value for Software Architectural Drivers for Cloud Testing obtained is 0,901 because QoP is 0,226 and QoS is 0,675.

Figures 5 and 6, suggested by the authors, show numerically the Quality of Products (QoP) and Quality of Services (QoS) for each Cloud Testing phases.

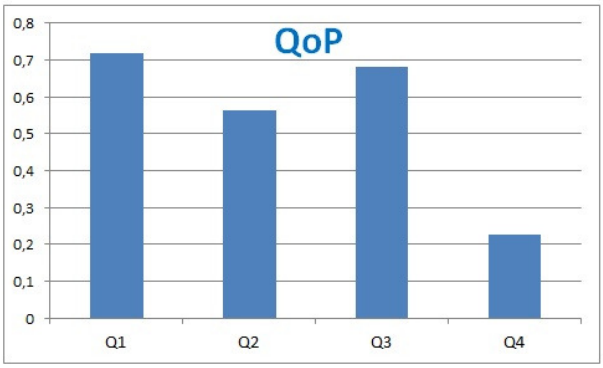


Figure 5. The QoP Results for each Cloud Testing phases.

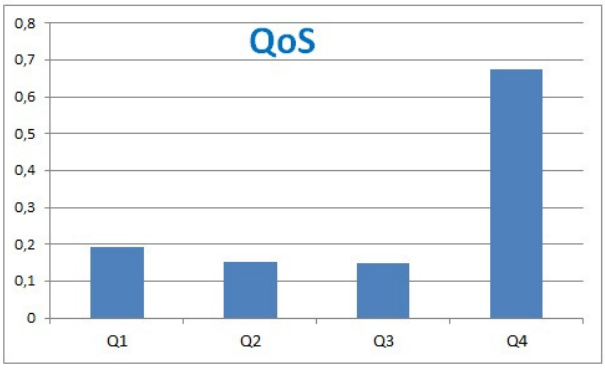


Figure 6. The QoS Results for each Cloud Testing phases.

IX. Conclusion and Future Work

This research has provided the investigation, design, and implementation of some key Software Architectural Drivers for Cloud Testing, focusing on monitoring the quality for both end products and services.

 ISBN: 978-1-61208-233-2

The Software Architectural Drivers for Cloud Testing, proposed by the authors, was evaluated through priorities and weights assigned to them, by seasoned testers. Other drivers could have been included and theirs effects measured for Cloud Testing. However, in this research, the proposed drivers have been proved appropriated, based on the above criteria.

The drivers, prioritized and weighted by experts, have allowed the quantitative monitoring of quality on Cloud Testing.

As a result of this research, it was obtained a way to numerically calculate the quality of Cloud Testing.

From this research, it is possible to evaluate the influence of each Software Architectural Drivers for Cloud Testing, by prioritizing and weighting each driver. It is also possible to measure the influence of each individual driver on the overall quality of Cloud Testing, by assigning it a numerical value.

Within the Cloud, the test must start earlier (in a very early stage); the test scope is widened because of its nonfunctional requirements; and the test must never stop (due to the fact that there are a lot of continuous services to be performed and also due to constant environment changes). This assures that the software is tested thoroughly.

The authors recommend the continuation of this research in the Cloud Production. A question that arises from this work is: "Software Architectural Drivers for Cloud Testing in production can be applied?"

The answer to this question can be obtained through further experiments.

As future works, it is suggested the application of these drivers into other experiments and a statistical in-depth evaluation about its effects on Cloud Testing.

This would foster better QoS, as end products, by fulfilling some existing gaps of knowledge within the Cloud Computing environment.

REFERENCES

[1] W. Jun and F. Meng, "Software Testing Based on Cloud Computing," International Conference on Internet Computing and Information Services, 2011.

[2] T. Parveen, and S. Tilley, "When to Migrate Software Testing to the Cloud?," In proc. 2nd International Workshop on Software Testing in the cloud (STITC), 3rd IEEE International Conference on Software Testing, Verification and Validation (ICST), April 2010, pp. 424-427.

[3] A.W.S. Team, "Summary of the Amazon EC2 and Amazon RDS Service Disruption," Amazon Web Services [Online]. Available: <http://aws.amazon.com/pt/message/65648/> 10.18.2012.

[4] B. Gallagher, "Using the Architecture Tradeoff Analysis Method to Evaluate a Reference Architecture: A Case Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Note CMU/SEI-2000-TN-007, 2000.

[5] B.Batke and P. Didier, "The importance of Reference Architecture in Manufacturing Networks," CIP Networks Conference, 2007. Available:<http://www.odva.org/Portals/0/Library/CIPConf_AGM/ODVA_12_AGM_The_Importance_of_Reference_Architectures_Didier_Batke.pdf> 10.18.2012.

[6] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," SP 800-145, National Institute of Standards and Technology's, U.S., 2011.

[7] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf, "NIST Cloud Computing Reference Architecture," SP 500-292, National Institute of Standards and Technology's, U.S., 2011.

[8] K. Blokland and J. Mengerink, "Cloutest®: Testen van cloudservices," Uitgeverij Tutein Nolthenius, 2012.

[9] R. Kazman, M. Klein, and P. Clements, "ATAM: Method for Architecture Evaluation," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-2000-TR-004, 2000.

[10] M. Veras, "Virtualização: Componente Central do Datacenter," Brasport, Sérgio Martins Oliveira, 2011.

[11] ABNT, "NBR ISO/IEC 25000. software quality requirements and evaluation," Associação Brasileira de Normas Técnicas, 2008.

[12] W. E. Perry, "Effective Methods for Software Testing," N.Y.: Wiley, 2006.

[13] M. Cohn, "Succeeding with Agile: Software Development Using Scrum," Addison-Wesley Professional, 2009.

[14] L. Crispin and J. Gregory, "Agile Testing: A Practical Guide for Testers and Agile Teams," Addison-Wesley Professional, 2009.

[15] IEEE Std 829-2008, "IEEE Standard for Software and System Test Documentation," Institute of Electrical and Eletronics Engineerd, 2008.

[16] W. Jansen and T. Grance, "Guidelines on Security and Privacy in Public Cloud Computing, " SP 800-144, National Institute of Standards and Technology's, U.S., 2011.

[17] L. Badger, T. Grance, R. Patt-Corner and J. Voas, "Cloud Computing Synopsis and Recommendations - SP800-146," National Institute of Standards and Technology's (NIST), 2012.

[18] N. Kannan, "Ten tests for software applications in the cloud," SearchCloudComputing, TechTarget, Inc. 275 Grove St. Newton, MA 02466, 2011.

[19] M. Laverick, "Private Cloud e-zine," vol. 1, SearchCloudComputing, TechTarget, Inc. 275 Grove St. Newton, MA 02466, 2011.

[20] J. D. McCaffrey, "Using the Multi-Attribute Global Inference of Quality (MAGIQ) Technique for Software Testing," Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on, 2009, pp. 738-742.

[21] J. Jia, G. W. Fischer and J. S. Dyer, "Attribute weighting methods and decision quality in the presence of response error: a simulation study," Journal of Behavioral Decision Making, 1998, vol. 11, no. 2, pp. 85-105.

 ISBN: 978-1-61208-233-2

Optical Link Testing and Parameters Tuning with a Test System Fully Integrated into FPGA

Anton Kuzmin, Dietmar Fey
Department of Computer Science,
Chair of Computer Architecture
Friedrich-Alexander-University Erlangen-Nuremberg, Germany
{anton.kuzmin,dietmar.fey}@informatik.uni-erlangen.de

***Abstract*—Development, characterization and performance optimization of systems utilizing FPGAs with high-speed serial transceivers to implement optical links with 1 to 10 Gbps data rate is a complex task and it poses several challenges for design engineers. In this paper, an effective approach is presented designed to address these challenges based on the use of diagnostic features implemented in the transceivers and a soft-IP microcontroller system instantiated in the FPGA. The use of the soft-IP controller allows a single-point access to the control and diagnostic interfaces of all components forming the link. Combined with computational capabilities and a high-level programming language interpreter running on the soft-IP CPU inside the FPGA, it enables extensive optical link performance evaluation without relying on any additional test and measurement equipment and significantly shortens debugging and testing times. The implementation demonstrates the feasibility and effectiveness of the proposed approach to utilization of on-chip diagnostic capabilities.**

***Index Terms*—Optical fiber communication; Transceivers; FPGA; Microcontrollers; Embedded software**

I. Introduction

Modern applications including rich media content transport, on-the-fly image processing, high bandwidth data acquisition for experimental physics, and high performance computing, require ever increasing serial communication data rates. At the same time, latency requirements remain strict and significantly limit possibilities for error correction and therefore call for a lower number of acceptable errors in the communication channel. FPGA devices with integrated high-speed serial transceivers and optical interconnects provide a very efficient and flexible platform for implementing such demanding applications and can be found in an increasing number of systems. Various examples and applications of optical interconnects could be found in [1]–[5].

One of the major challenges is a parameter tuning of the various components forming an interconnect to achieve the lowest possible probability of bit errors. The problem is that accurate measurements at low error probabilities require very long times even at high data rates to accumulate statistics for a given confidence level while the parameter optimization space is relatively big. Additional complications arise from the fact that various components of the link have very different interfaces for setting parameters. In most cases, they are supported by proprietary tools with limited functionality for automatically tuning link parameters. The application of these tools often requires a connection of the system to external test and measurement equipment. The limitations associated with its usage become increasingly severe with a tighter integration between the FPGA and the optical transceiver blocks as recently proposed by Li et al. [6]. This level of integration makes electrical signals between the FPGA and optical transceiver practically inaccessible for external test equipment.

This paper presents an effective approach designed to address challenges associated with the testing, parameter tuning and performance monitoring of optical interconnects in FPGA-based systems. The approach is based on the use of a soft-IP controller embedded into the FPGA to perform two major tasks: link performance measurements and control of parameters of the different components forming the link.

The paper has the following structure. In the first section, an example of utilization of FPGA built-in transceiver diagnostic capabilities is presented and the key differences in the approach chosen by the authors are outlined. In the subsequent section an overall inter-FPGA transceiver-based serial link structure is shown followed by brief description of its components and their respective configurable and tunable parameters. Then, a Bit Error Ratio (BER) [7] is introduced as an integral characteristic of link performance. An optimized algorithm for obtaining an accurate BER scan plot (bath-tub curve) is described. It can be used for indirect eye diagram width measurement by introducing a phase shift into a signal sampling point inside the receiver. The eye diagram width may serve as an indicator of the link performance and is used as a target function for the link parameter optimization.

Implementation aspects of the FPGA-based optical link test system are then discussed in the next parts of the paper along with the obtained link performance measurement results. Comparison of the measured BER levels for different optical modules confirms the validity of the implemented test system.

The factors limiting a wider adoption of the approach presented, possible ways to address them and directions for further research and development work are discussed in the concluding section.

II. Related Work

Usage of FPGA for testing communication channels has been previously described. For instance, in [5] an implementa-

 ISBN: 978-1-61208-233-2

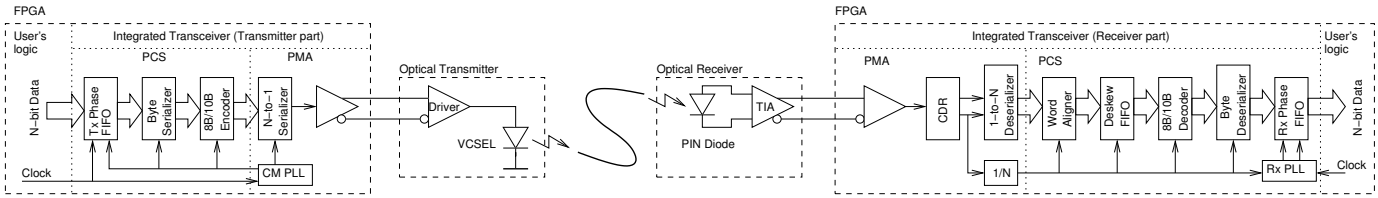


Fig. 1. Simplified inter-FPGA serial optical link structure.

tion of the Bit Error Ratio Tester (BERT) based on the Alera's Stratix II GX transceivers is presented and compared with a commercial stand alone tester. It is shown that the results obtained with the FPGA implementation comply with the results of the stand alone tester. However, the implementation still utilizes external equipment to control the test system and to collect the measurement results.

This paper, while similar in overall approach to the one proposed by Xiang et al. [5], presents notable improvements in several areas. The most important of these is the implementation of a functionally complete test system inside the FPGA. Additionally the flexibility of the implemented system allows extension of its hardware and software components to support interfaces for monitoring and controlling the parameters of the various components of the link without external equipment. Another improvement presented in this paper is an adaptation of eye-width as a link performance indicator instead of a raw BER. The eye-width can be measured significantly faster at low bit error probabilities with the aid of diagnostic circuitries integrated into the transceivers and therefore is more efficient as a target function for the parameter space exploration and link performance optimization.

III. Optical Link Structure

A block diagram of an optical digital communication link is shown in Figure 1. The link data path consists of a transmitter, an electro-optical converter (VCSEL with its driving circuits), an optical fiber, a photo detector (PIN diode and transimpedance amplifier) and a receiver. The transmitter and receiver are further divided into a Physical Coding Sublayer (PCS) and a Physical Medium Attachment (PMA) sublayer.

The PCS blocks are responsible for byte serialization/deserialization, byte ordering, rate matching, and 8B/10B encoding/decoding. All these functions are essential for the implementation of a reliable digital data channel. However, in this work, we concentrate on the physical layer performance measurements leaving the problems related to the coding sublayer out of the scope of the research.

The transmitter part of the transceivers integrated into the FPGA allows the tuning and run-time changes of several parameters. Among them are clock multiplication phase-locked loop (PLL) dividers and bandwidth, output driver common mode voltage, differential voltage output swing and preemphasis aimed at reducing the negative effects of inter-symbol interference. The receiver part, in turn, has the following tunable blocks and parameters: on-chip termination, adaptive equalization, decision feedback equalization, receiver input common mode voltage and gain. These blocks have a crucial impact on the signal quality on the input of the Clock and Data Recovery (CDR) circuitry, but their influence cannot be measured directly because the signal after these stages is not physically available outside the chip and cannot be connected to external measurement equipment. The CDR block provides a built-in diagnostic support circuitry to facilitate assessment of the signal quality on its input.

The hardware interfaces, which are necessary to change all the transceiver's parameters and to access the diagnostic circuits, are available to the logic programmed into the FPGA. Chip and design software vendors provide tools to access these interfaces, however their use requires a connection between the development workstation with CAD software and the FPGA. The electro-optical components of the link have their own sets of tunable and monitoring parameters, such as driver and receiver power levels, VCSEL modulation and offset currents, temperatures and thermal compensation coefficients, signal power detected at the receiver input, etc. Access to these features is implemented through another set of vendor-specific interfaces and also requires a development workstation with a connection to the target system. Such connections may be not feasible in the embedded system while access to the interfaces is still highly desirable or even required. This problem may be addressed by integration of IP cores for all required management interfaces into the system instantiated in the FPGA.

The flexibility of a soft-IP microcontroller system inside the FPGA allows the implementation of a single-point access to the management interfaces of all the components forming the link. Combined with built-in link diagnostic capabilities controlled by the same microcontroller system it results in a complete test system that enables link performance testing and parameter tuning without relying on any external equipment. Additionally it is available not only during development and testing of the system but also after its deployment.

IV. Link Performance Indicators

Two link operation quality indicators are introduced in this section along with a description of an algorithm used by the authors to measure "eye-width" with the transceiver's built-in diagnostic circuits.

A. Bit Error Ratio

The integral quality of operation of a serial link is characterized by its Bit Error Ratio (BER): a ratio of the number of bits received with errors to the total number of bits

 ISBN: 978-1-61208-233-2

transmitted through the link: $BER = N_{err}/N$. This ratio is used for both measured and actual values. A BER is usually measured with a special piece of test equipment, so called Bit Error Ratio Tester. It consists of a data pattern generator, a reference quality receiver, a digital comparator and counters for transmitted bits and errors. The flexibility of an FPGA allows to implement all blocks of a bit error ratio tester in programmable logic in the FPGA itself.

The measured value approaches the actual BER in the limit: $\lim_{N\to\infty} N_{err}/N = p_e$. It is not possible for BER measurement to transmit an infinite number of bits since it would require an infinite measurement time and a way to measure the BER with a given accuracy is required. For practical application it is often enough to know that the BER is below some threshold with a given confidence while its actual value is irrelevant. As the literature shows (for instance, in [8]), if more than N_0 bits were transfered during the test with no errors detected, then with probability α the actual BER is less than p_e:

$$N \geq N_0 = \frac{1}{p_e} \ln \frac{1}{1-\alpha}$$

This number of bits (N_0) sets a lower limit on the test duration when no errors are observed. At a data rate of 5 Gbps it takes approximately 10 minutes to reach a 95% confidence that BER is lower than 10^{-12}, for the BER level of 10^{-15} it would require almost a week. The long runtime required makes it impractical to use the BER directly as a target function for the link parameters optimization. It would take enormous amount of time to find an optimum in the parameter space even if only a small fraction of all possible parameter combinations yielded a bit error ratio lower than 10^{-12}.

B. Eye-Width and its Measurement

The quality of a signal may be analyzed by evaluating its eye diagram: a picture on an oscilloscope display resulting from observing a transmission of a pseudo-random binary sequence with properties representative of the physical layer encoding used in the link. The width and height of an opening of the central part of the diagram ("eye") serve as indicators of the signal quality and may be used as target functions for the link parameter tuning. However, the signal on the input of the receiver CDR unit is not available for direct measurements. Therefore built-in diagnostic circuitries of the receiver should be utilized.

Serial transceivers integrated into the Altera Stratix IV GX FPGAs include special circuitry that facilitates measurements of the eye opening on the input of the CDR block [9]. The circuitry allows shifting of a sampling point of the signal from its optimal position in the center of the unit interval (UI) under external control. Then bit error ratio is measured for each phase offset. For sampling points close to the center of the eye opening, there will be no significant increase in the bit error ratio. For sampling points closer to the signal slopes the number of observed errors will gradually increase. Finally, in the area of the signal edge crossing widened by a jitter, a receiver will not be able to achieve synchronization with its input signal resulting in the observed bit error ratio of 0.5. From these measurements of the BER at signal sampling points distributed through the UI the eye opening and jitter characteristics of the signal may be deduced [8].

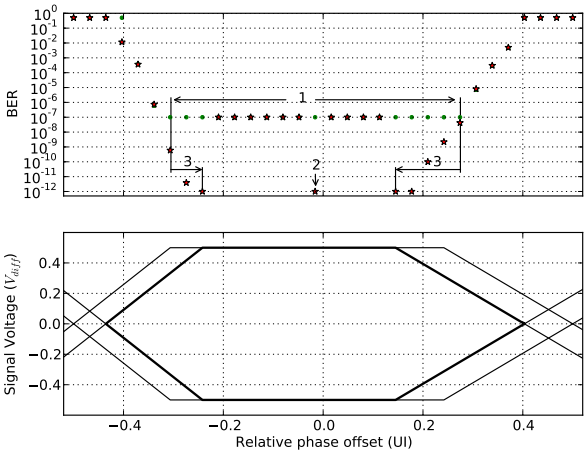


Fig. 2. "Bath-tub" curve scan algorithm and reconstructed eye diagram.

The key benefit of this approach is that the conclusion regarding the signal quality and, therefore, link parameters, may be reached by a number of BER measurements with different phase offsets through the UI instead of one at the optimal sampling point. However, each of these measurements needs to achieve a given confidence level at a much higher target BER and, therefore, requires significantly shorter runtime.

An algorithm implementing this approach can be further optimized to reduce the number of required BER measurements at the center of the eye opening, where the bit error ratio is low. These measurements take up most of the time and effectively provide no useful information. Several approaches to such optimization are described in [8].

Figure 2 illustrates the behavior of the modified algorithm implemented by the authors and shows an eye-diagram reconstructed from the measurements. As a first step (marked with 1 in the figure) an initial scan through the entire unit interval is performed with high target BER (10^{-7}). From these measurements, an approximate location of the eye boundaries is determined. At the second stage the BER is measured at the center of the eye opening to make sure that the target BER level (10^{-12}) is achievable at the close-to-optimal sampling point (2). Then, the BER is measured at sample points from the eye opening boundaries detected during the first scan towards the center to determine points where the target BER level is achieved (3). The distance between these points (eye-width) serves as a measure of the signal quality at the input of the receiver CDR unit and may be used as a target function for the link parameters' tuning.

The described algorithm for eye-width measurements reduces the number of BER samplings within the eye opening. For the diagram shown in Figure 2, it took only 55 minutes

 ISBN: 978-1-61208-233-2

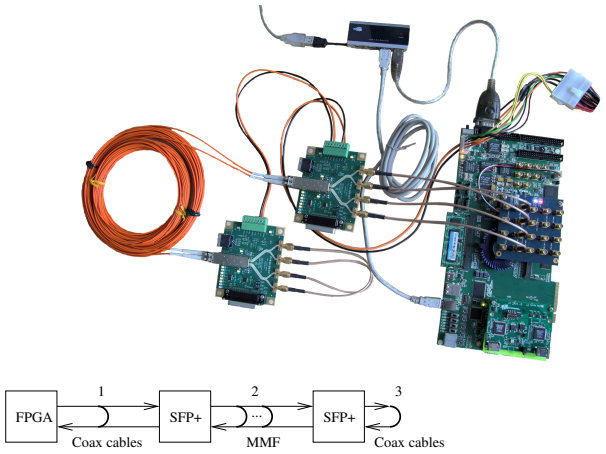


Fig. 3. Experimental system and loopback configurations.

to collect all the data. An exhaustive UI scan under the same conditions takes 150 minutes but provides no additional information on the link operation.

V. Test System Implementation

To confirm the usefulness of the approach described to the optical link testing and parameter tuning and to create a base set of tools to be used in future projects, e.g., in an FPGA-based HPC system exploiting high speed optical interconnects, the authors implemented a prototype system. The system consists of hardware, a set of IP blocks, embedded software and development tools and facilitates debugging, testing and evaluation of the components. A photo of the assembled system hardware is shown in Figure 3 and components of the system are described in the following sections.

A. Hardware Platform

The system is based on the Altera Stratix IV GX FPGA (EP4SGX230KF40C2) installed on a TerasIC DE4 board. Through an adapter board with SMA connectors and a set of coaxial cables the DE4 board is connected to SFP+ evaluation boards hosting optical transceiver modules. Hot-pluggable SFP+ transceivers used in the system provide duplex LC-type optical connectors for the Multi-Mode Fiber. Management interface of the transceiver modules (I^2C) is accessible from the FPGA and is used for the monitoring of their parameters.

The highly modular construction of the hardware platform enables experimentation with different components and link configurations. During development and validation of the system several loopback configurations were used as shown in the diagram on Figure 3. The shortest possible one is an electrical loopback connecting the FPGA transmitter output signals directly to the input of the receiver (1). The second tested configuration uses a single optical transceiver with its input and output connected via a Multi-Mode Fiber (MMF) loopback (2). The length of the fiber loop used in the tests ranged from 15 cm to 15 meters. This loopback configuration is the closest to an actual optical link where the signal passes through one electro-optical and one opto-electrical conversion and a single fiber segment.

The most elaborate loopback configuration tested utilizes two transceiver modules and an electrical loopback on the "remote" side of a duplex fiber link (3). While this link exceeds configurations, which would be found in practical applications it is still interesting as it allows an easier separation of influence on the signal quality from different components of the link and serves as a model of a less favorable environment with longer links and a higher number of interconnects along the signal path.

The transceivers available in Stratix IV GX FPGA provide an on-die scope capable of $1/32$ unit interval resolution at data rates up to 6.5 Gbps [9]. Comparable technology is available in the transceivers integrated into the Xilinx Virtex-6 FPGA family. As an additional feature these transceivers are capable of a vertical scan of an eye-diagram [10], however this functionality has not yet been explored by the authors so far.

B. System-on-Programmable Chip and IP Cores

The architecture of a soft-IP microcontroller system instantiated in the FPGA is shown in Figure 4. The system consists of the following main blocks: NIOS II CPU core with a small on-chip ROM containing boot code, a controller for external SRAM and FLASH, UART for communication with a control terminal, cores for the test pattern generator and checker, interfaces to access the transceiver configuration and diagnostic features, I^2C master cores for connection to the management interface of the SFP+ modules. The entire system utilizes only a small fraction of the available FPGA resources: the logic utilization is 3%, and available memory and DSP blocks are used for less than 1%.

The IP cores forming the system were taken from three sources. The first one is the library supplied by the FPGA vendor (Altera in this case). The cores are optimized for a specific FPGA architecture, but no source code is provided and the cores are not available on FPGAs from other vendors. The second source of IP cores for the system is a collection of free and open cores hosted on the OpenCores site [11]. These cores are provided under free licenses and their source code is available. This makes it possible to implement these cores in systems on different FPGA architectures. The price for such flexibility is the time and effort required for integration and adaptation, and the required time and effort is generally greater than for FPGA vendor supplied IP cores.

These two sources of IP cores, while covering most of the functionality, still do not provide several crucial interfaces required in order to access transceiver configuration and diagnostic interfaces. These missing parts were created by the authors by means of custom HDL development as the third source of IP blocks, and this required most effort.

Since the IP cores from different sources have different interfaces their integration into a working system is a technical problem in itself and required the development of "adapter"

 ISBN: 978-1-61208-233-2

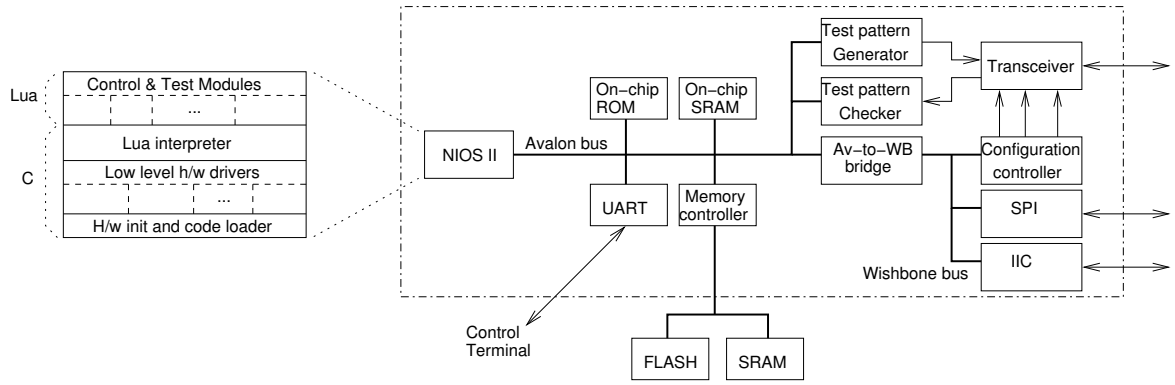


Fig. 4. Test System-on-Programmable Chip (SoPC) architecture.

modules. The two primary on-chip interconnects used in the system are Avalon [12] and WISHBONE [13].

Overall, a combination of the readily available blocks (both proprietary and free) and those developed in-house proved to provide a reasonable and time efficient way of implementing the prototype system.

C. Embedded Software

The monitoring and control of all blocks forming the optical link, BER testing and processing of the test results are handled by an embedded software running on the NIOS II soft-IP CPU instantiated in the FPGA.

Low level software to access all hardware interfaces is implemented in the C programming language and its functionality is made available to the Lua interpreter. Lua, as is stated on its web-site [14], "is a powerful, fast, lightweight, embeddable scripting language". These properties make it very attractive for a wide range of applications including game development, mobile devices and embedded software [15]. A tight integration with C and an interactive interpreter facilitate an efficient development of diagnostic, testing and debugging software for embedded hardware systems.

Availability of an ANSI C compiler and a basic C run-time library are the only requirements to port Lua to a new platform and it was extremely easy to get an early prototype running on NIOS II. The efforts invested in the porting and support of Lua interpreter on the soft-IP microcontroller system in the FPGA were rewarded in the flexibility of the resulting system and increased development productivity.

Access to the interactive environment is very useful during embedded hardware development and debugging as it saves a lot of time in the edit-compile-load-run development cycle. Since the "hardware" itself is a soft-IP system instantiated in the FPGA this time saving becomes even more important: on the one hand, the system is malleable and experimental and includes design errors, on the other hand traditional software development cycle is complicated by a separate FPGA design flow with longer iterations. With this additional complexity an availability of tools facilitating quick experiments and tests running directly on the target platform is a key factor for effective development. Our experience shows that Lua fits this role perfectly and allows rapid localization of the design errors both on the hardware and software levels. All the link configuration and BER measurement software in the system are implemented as a set of Lua modules.

VI. MEASUREMENT RESULTS

Measurements on the test system were performed for data rates in a range from 1 to 5 Gbps with various loopback configurations. The SPF+ module used in most experiments is the Avago AFBR-703SDDZ. The module is capable of data rates up to 10 Gbps and, as expected, performs excellently in the tested data rate range. Even with the most demanding loopback configuration the eye diagram opening for the 10^{-12} BER level is approximately 40% (80 ps) of the unit interval (200 ps at 5 Gbps).

Several data patterns with different spectral characteristics were used in the experiments. Two test patterns that specifically check the link performance at the edges of its frequency band are the Low Frequency (LF) and High Frequency (HF) patterns. The other test patterns are Pseudo-Random Binary Sequences (PRBSx) generated by a linear feedback shift register with the length x. The lengths of 7, 15, 23, and 31 bit were used. The test results show slight dependency on the data pattern used, however detailed analyses of this dependency have not yet been yet performed.

To validate the test system and confirm that the measurement results adequately represent link quality an SFP module with a lower maximum data rate has been used: Finisar FTLF8524P2BNL. According to its documentation the module is capable of data rates up to 4.25 Gbps. Experiments show that up to this limit it demonstrates BER $\leq 10^{-12}$, also the eye width is smaller than that with the Avago module. The bathtub scan results for both modules at 5 Gbps are shown in the Figure 5. This data rate is outside of the specified range for the Finisar module and this is clearly visible from the diagram: even in the vicinity of the ideal sampling point BER does not achieve 10^{-7} level.

The results obtained allow the conclusion that the developed test system provides reliable data on the optical link performance and may be used to compare different link implementations and to tune parameters of the link. The comparison of

 ISBN: 978-1-61208-233-2

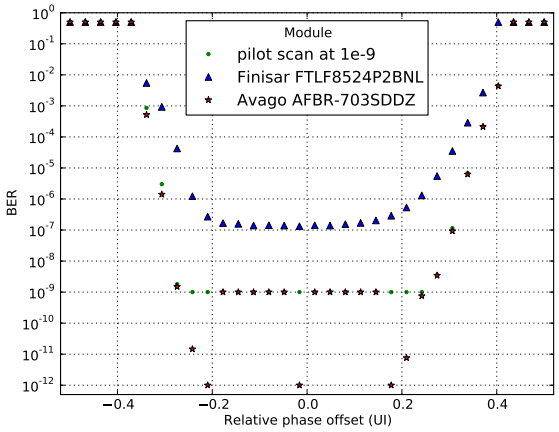


Fig. 5. Comparison of "bath-tub" curves for two SFP modules at 5 Gbps.

the measurement results obtained with different data patterns may provide additional information that could be useful for optimizing link performance.

VII. Conclusion and Future Work

The implementation clearly demonstrated the feasibility and effectiveness of the proposed approach to utilization of the on-chip diagnostic capabilities of FPGAs with high-speed serial transceivers. The use of the soft-IP controller instantiated in the FPGA allows a single-point access to the control and diagnostic interfaces of all components forming the link. Combined with computational capabilities and a high-level programming language interpreter running inside the FPGA, it enables extensive optical link performance evaluation without relying on any additional test and measurement equipment and significantly shortens the system debugging and testing times. As an additional benefit all the implemented functionality is still available in the deployed system and may be used for remote monitoring and diagnostics.

Several factors limit a wider application of this approach. One of the most critical is the utilization of FPGA vendor specific IP cores. To use the test system on an FPGA from a different vendor these blocks should be replaced with their functional equivalents available on the other platform, but supporting different system variants would increase the effort required. A more efficient approach is to replace the vendor specific IP cores with free and open-source equivalents available on all target platforms.

The most complex and important block in the system specific to the Altera platform is the NIOS II CPU core and its replacement with one of the free CPU cores is considered by the authors to be the next step in the project. The remaining proprietary cores (test data pattern generator and checker, external bus controller, UART) are expected to be easier to replace and do not require toolchain and embedded software porting effort. The replacement of the IP blocks available only on one FPGA architecture with portable ones will make it possible to reuse the test system on different FPGAs and boards and will facilitate direct comparison of the optical modules and built in FPGA transceivers across them.

Another area for improvement is the automated integration of separate IP blocks from different sources into a system. Vendor specific tools have progressed notably in this area in recent years, however they are still limited with regard to support of "foreign" IP cores. On the other hand, while efforts have being made to provide similar functionality for free and open-source cores, the tools that have emerged so far are not well integrated in the FPGA and embedded software design flows.

Detailed analysis of the dependencies between the test loopback configurations, data patterns, transceiver parameters and observed eye-diagram is required to develop effective algorithms for link parameters tuning. This work provides efficient tools for these researches and demonstrates their feasibility.

The listed tasks are aimed at improving the implemented test system itself. The other step planned is to apply the system to the characterization and parameter optimization of the 12-channel parallel optical links built with IPtronics low-power VCSEL driver and TIA arrays or emerging MiniPOD optical modules. The developed blocks are planned to be used in a reconfigurable research HPC system with optical interconnects currently under development.

References

[1] A. F. Benner, M. Ignatowski, J. A. Kash, D. M. Kuchta, and M. B. Ritter, "Exploitation of optical interconnects in future server architectures," *IBM Journal of Research & Development*, vol. 49, no. 4/5, p. 755, July/September 2005.

[2] S. Nakagawa, Y. Taira, H. Numata, K. Kobayashi, K. Terada, and M. Fukui, "High-Bandwidth, Chip-Based Optical Interconnects on Waveguide-Integrated SLC for Optical Off-Chip I/O," in *Electronic Components and Technology Conference*, 2009, pp. 2086–2091.

[3] B. E. Lemoff, M. E. Ali, G. Panotopoulos, E. de Groot, G. M. Flower, G. H. Rankin, A. J. Schmit, K. D. Djordjev, M. R. T. Tan, W. Gong, R. P. Tella, B. Law, and D. W. Dolfi, "Parallel-WDM for multi-Tb/s optical interconnects," in *Lasers and Electro-Optics Society (LEOS) IEEE Meeting*. Agilent Technologies Laboratories, 2005, pp. 359–360.

[4] O. Liboiron-Ladouceur, H. Wang, A. S. Garg, and K. Bergman, "Low-Power, Transparent Optical Network Interface for High Bandwidth Off-Chip Interconnects," *Optics Express*, vol. 17, pp. 6550–6561, 2009.

[5] A. C. Xiang, T. Cao, D. Gong, S. Hou, C. Liu, T. Liu, D.-S. Su, P.-K. Teng, and J. Ye, "High-Speed Serial Optical Link Test Bench Using FPGA with Embedded Transceivers," in *Topical Workshop on Electronics for Particle Physics (TWEPP)*, 2009, pp. 471–475.

[6] M. P. Li, J. Martinez, and D. Vaughan. Transferring High-Speed Data over Long Distances with Combined FPGA and Multichannel Optical Modules. [Online]. Available: http://www.altera.com/literature/wp/wp-01177-AV02-3383EN-optical-module.pdf [retrieved: March, 2012].

[7] G. Breed, "Bit Error Rate: Fundamental Concepts and Measurement Issues," *High Frequency Electronics*, pp. 46,48, January 2003.

[8] M. Müller, R. Stephens, and R. McHugh, "Total Jitter Measurement at Low Probability Levels, Using Optimized BERT Scan Method," in *DesignCon*. Agilent Technologies, 2005.

[9] W. Ding, M. Pan, T. Tran, W. Wong, S. Shumarayev, M. Peng Li, and D. Chow, "An On-Die Scope Based on a 40-nm Process FPGA Transceiver," in *DesignCon*. Altera Corporation, 2010.

[10] *RocketIO Transceiver User Guide*, Xilinx, Inc., 2007.

[11] [Online]. Available: http://opencores.org/ [retrieved: October, 2012].

[12] Avalon interface specifications. [Online]. Available: http://www.altera.com/literature/manual/mnl_avalon_spec.pdf [retrieved: May, 2011].

[13] Wishbone B4. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. [Online]. Available: http://cdn.opencores.org/downloads/wbspec_b4.pdf [retrieved: October, 2012].

[14] [Online]. Available: http://www.lua.org/ [retrieved: October, 2012].

[15] R. Ierusalimschy, *Programming in Lua*. Lua.org, 2006.

 ISBN: 978-1-61208-233-2

Data Model Based Test Case Design

Model-driven Information System Testing

Federico Toledo Rodríguez
Abstracta
Montevideo, Uruguay
e-mail: ftoledo@abstracta.com.uy

Beatriz Pérez Lamancha
Software Testing Center,
University of the Republic,
Montevideo, Uruguay
e-mail: bperez@fing.edu.uy

Macario Polo Usaoloa
Alarcos Research Group,
University of Castilla-La Mancha,
Ciudad Real, Spain
e-mail: macario.polo@uclm.es

***Abstract*—Software testing is a challenging task, but frequently the time is wasted in interactions between development team and testing team due to simple errors related with the data structure and neither with the complex business rules. That highlights that it is very important to verify that the application can handle correctly the data structure and the data types, and for this we consider to generate test cases based on the data model. We are developing a framework to generate executable test cases from a data model, to test information systems that use databases. In this article, we will present the test case design approach, based on the data model, in order to verify the correctness of the application layers that manage it.**

Keywords-test data; information system testing; model driven testing; automated test case generation

I. INTRODUCTION

The design of many applications starts with a conceptual modeling which is then used to define the database schema and the classes' structure of the domain tier of the application to be developed. Domain classes are then enriched with both methods to deal with the business goals, and with methods to deal with the persistence of their instances. Considering that the database structure is well designed, according to the requirements and the performance needs, then, it is necessary to verify that the application layer over it can manage correctly the particularities of the defined structure. Moreover, the same database structure could be accessed by different applications, such as a Web application for the customers, a desktop application as a backend, or a layer exposing Web Services in order to provide an integration mechanism with other systems. Thus, there is a correspondence between the logic components (e.g. classes, servlets and services) and the data structures (generally in a relational database). As the basic operations to manipulate data structures are the CRUD operations (*create*, *read*, *update*, *delete*) and almost any business method changing the state of a persistent instance will do a call to a CRUD operation, we will pay special attention on these methods on each entity.

Model-Driven Testing (MDT) [1] implies the automatic test case generation from models through model transformation. Our methodology follows a model-driven testing approach to automatically generate test cases from the data model, obtained from the database metadata. The generated test cases permit to verify the correctness of the CRUD operations of the entities defined in the system, according with certain coverage criteria. The methodology is supported with a framework that is based in the most important standards, mainly in the Unified Modeling Language (UML) [2].

In this article, we present how we design the test cases for information systems with databases. In Section II, the general framework is introduced. Then, in Section III, we present the main contribution of this article which is the test case design strategy. Section IV shows the state of the art regarding with test cases generation for database-driven applications. Finally, Section V draws some conclusions and future lines of work.

II. FRAMEWORK FOR INFORMATION SYSTEM TESTING

The methodology has three main phases (*Figure 1*). Each step we fits into different standards mainly from the *Object Management Group* (OMG), especially UML, in order to use general UML modeling tools. These three phases are:

- **Phase 1: Reverse Engineering.** Initially some reverse engineering techniques and tools are used in order to obtain the corresponding data model, from the physical schema of the database.
- **Phase 2: Model to Model Transformation.** The data model is processed looking for certain patterns and then generating automatically test cases for each pattern through model transformations. As a result, test cases for the data structures are generated, thus obtaining a test model.
- **Phase 3: Model to Text Transformation.** Last but not least, the test models are transformed into test code, obtaining executable test cases.

In order to represent the data model we use the UML Data Modeling Profile (**UDMP)** [3], that is an UML class diagram extension developed by IBM to design databases using UML, with the expressive power of an entity-relationship model. It defines concepts at a physical level and architecture (*Node, Tablespace, Database,* etc.), and the ones required for the database design (*Table, Column,* etc.). Several proposals use this profile to model the database structure [4-6].

 ISBN: 978-1-61208-233-2

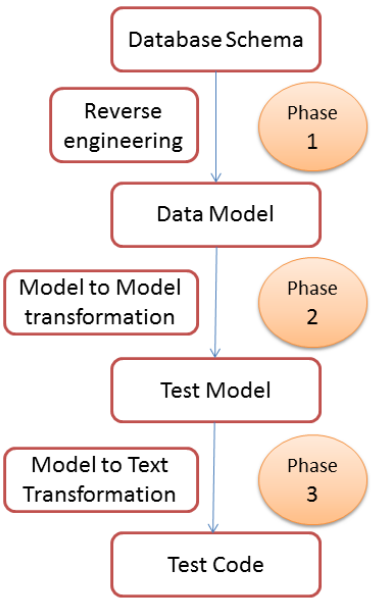


Figure 1 - Methodology and framework

For more detail on the framework, refer to [7]. In the following section, we focus on the test case design that is the most important part of the design of the second phase.

III. DATA MODEL CENTERED DESIGN

Given that in our case we generate test cases from a UDMP class diagram corresponding to the system data model, we will consider some coverage criteria as adequate to those artifacts, as for example some of the proposed by Andrews et al. [8] UML class diagrams:

- **Class Attribute (CA):** the test suite should make use representative values for each attribute in each class.
- **Association end Multiplicity (AEM)**: the test suite should make use every representative pair of multiplicities for the associations of the model.

These coverage criteria were designed for the context of testing a method or use case, where an object oriented model defines the behavior of the system. In our case we will apply the criteria for a data model instead of an object model, so we adjusted some aspects in order to make it applicable. The most important consideration is that the operations that we will be testing are *create*, *read*, *update* and *delete* of each entity. This is important to determine the oracle, because the expected results of these operations are well-known. Another consideration related with the multiplicity of the associations: according with the definitions given in the foreign keys we could have different kinds of association multiplicities, and for each one we have to considerate a special situation about the boundaries of the association end multiplicities.

To apply these criteria the framework will generate test cases to cover these situations for every substructure of the data model that matches any of the criterion, what means that for each class it will generate test cases according with *CA* criterion and for each association will generate test cases according with *AEM* criterion.

We designed the patterns, and the corresponding test cases to be generated, according with the characteristics of the relations and tables involved. In the rest of this section we present an initial design for patterns with one table, two tables and three, describing the different situations and the test cases that will be generated in order to reach the defined coverage criteria.

A. *One-table Patterns*

First, we designed test cases to test the most basic patterns: based on one table, which means to pay special attention to the attributes and the different combinations of their representative values, according to *CA* criterion.

For each attribute we can categorize in valid data and invalid data, according with the data type obtained from the column metadata, and from business rules (extracted for example from the *Check* constraints defined in the database). This way, we are defining representative test data for each attribute. In this step, we define categories and values for each one, even considering boundaries. For instance, according with the example of the *Figure 2* (one table to store the name, id and age of people), the table *Persons* has an integer attribute age, and imagine that it is defined a *check* that verifies that the value is greater than zero, then, a set of interesting values could be: {-100, -1, 0, 1, 100}. Another interesting example is related with *varchar* variables, as the *id* attribute of *Persons*, as it is defined with a length of 50, we could try with a string with 50 or fewer characters, and one with more.

Once we have interesting values for each column, we combine them with pair-wise algorithms, using our own tool called CTWeb [9]. By this way we obtain a reduced set of tuples with higher probability to find errors. If we take the Cartesian product of the different interesting values, as suggested by Andrews et al. for the CA criteria, we will have too many values, so, we decided to reduce the test set by this way.

If any of the different attributes' values used by the test case is invalid, the expected result is a fail. If we test the *create* operation then we have to check that the instance was not created, and if all the values were valid, the expected result is a pass, and we should check that the instance was created correctly with the values used in the parameters. The

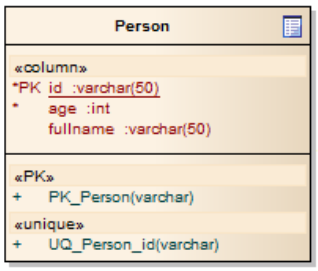


Figure 2 – Example with one table

same with the *update* operation, if all the input values are valid, we have to check that the values were updated, and if

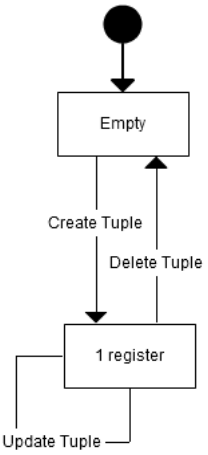


Figure 3 - State machine for 1 table

any of the input values were invalid, we have to check that the operation failed, and all the attributes in the database keep their original value.

The interesting operations are *create*, *read*, *update* and *delete* for each entity. The *read* operations are used to give support to the validation actions: if any assert fail, the error could be in the tested operation or in the *read* operation. Regarding *update* operation, there will be one for each attribute. Taking into account the previous considerations, we will apply the CRUD pattern [10] to considerate the whole life cycle of an instance, which implies to test the operation in the sequences that can be obtained expanding the regular expression: $C \cdot R \cdot (Ui \cdot R)^* \cdot D \cdot R$, where the *Ui* represents each operation that updates a different attribute. This is equivalent to generate test sequences according to the state machine presented in *Figure 3*, where there is an invocation to *read* operation in each state, in order to verify that the actual state is the expected.

Applying the CRUD pattern to the example of the entity *Person* we could generate the following test sequence:

1. Create Person
2. Read Person
3. Update Id Person
4. Read Person
5. Update Age Person
6. Read Person
7. Update Full Name Person
8. Delete Person
9. Read Person / should fail

Note that the final state of the database is the same one than the initial, what is convenient in order to have independent test cases: the execution order does not affect the expected result.

The same test sequence, which is the test behavior, can be executed with different test data, that it is going to be stored in a separated structure of the test model called *data pool*. This is known as *data-driven testing* approach [11], and the main advantage is that we can add easily new test cases just adding new rows to the data pool, indicating new interesting situations to cover with the data inputs. Therefore, the data pool will have the combination of the representing values, obtained from CTWeb.

B. *Two-table Patterns*

For this pattern, we will show as an example the one of *Figure 4*: the table *Journal* stores the different journals relating the editor responsible, whose information is stored in the table *Persons.*

Regarding the data inputs, we apply in each table the same process that for one table, except for those attributes included in the foreign key: first we define representative values and then we combine them with CTWeb in order to fill the data pools. For the foreign keys, we will have into account the *AEM* criterion, what means that we will try to associate instances in a way that covers the different representative multiplicities. The association ends of two tables (a referencing and a referenced table) could have multiplicity of 0..1 (in the referenced table side if the foreign key allows nulls, or in the referencing table side if the foreign key is unique), 1 (in the referenced table side if the foreign key does not allow nulls) or 0..* (in the side of the referencing table). Therefore, we can have the following combinations:

- 0..1 → 0..1
- 0..1 → 1
- 0..* → 0..1
- 0..* → 1

We are only considering the ones that can be implemented in a database schema with foreign keys, because for example the relation 1 → 1 it is not possible to implement with foreign keys between two tables.

The example of *Figure 4* corresponds with the last situation: 0..* → 1, from *Journal* to *Person*.

For each situation, we want to cover *AEM* criterion, and for this it is necessary to test associating entities with representative multiplicities, what is the boundaries of the defined ranges. For this, we consider to try each instance associated with 0, 1 and 2 instances of the other table. We consider that associating two instances is good enough to test the multiplicity "*".

According with this idea, different states of the database are defined, and considering the example of *Figure 4* some of these states are:

- One journal referencing one person (rel.: 1 – 1)
- Two journals with the same person (rel.: 2 – 1)
- One person that is not referenced (rel.: 1 – 0)

As we have 3 possibilities (0, 1 and 2) for each association end, we have 9 combinations. Some of these

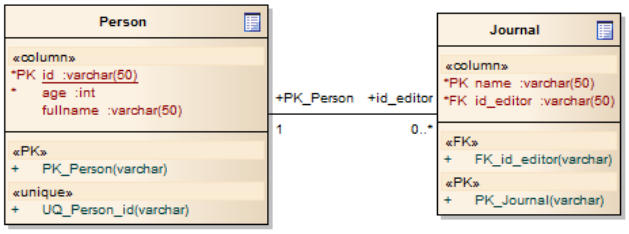


Figure 4 – Example with two tables

 ISBN: 978-1-61208-233-2

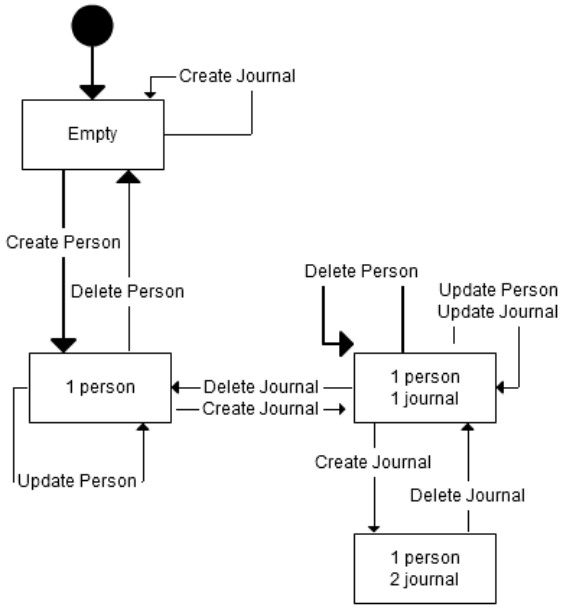


Figure 5 - Excerpt of the State Machine for Journal and Person

combinations are invalid according with the relation, as for example: in a relation 0..1 → 0..1, we cannot associate 2 registers with the same register of the other table. So, the expected result is defined by the validity of the data inputs and the validity of the number of instances to associate according with the foreign key.

The operations of *create*, *update* and *delete* force a change in the database state, only when we execute them with valid data. If we execute them with invalid data then the state should not change. The *update* operation also includes the update of the foreign key, considering that the valid data is the existing keys in the referenced table and invalid data when it does not, and similarly for *create* operation (it is interesting to test the creation of a *Journal* which references a *Person* that does not exist). If the foreign key has more than one attribute, it is necessary to considerate the update of them in the same operation.

With all these considerations, we defined a state machine, with the different states and transitions already described. The test cases that we design for this kind of patterns are based on the state machine coverage, for example trying to reach all paths, or all states and transitions. *Figure 5* shows an excerpt of the state machine for the example with *Journals* and *Persons*, and from this excerpt we present a possible test sequence generated from it (remember that after each operation there is a *Read* to verify the expected state):

1. Create Journal (without association) / should fail
2. Read Journal
3. Create Person
4. Read Person
5. Update Person (for each attribute)
6. Read Person
7. Create Journal with Person
8. Read Journal
9. Update Journal (for each attribute)
10. Read Journal
11. Update Person (for each attribute)
12. Read Person
13. Create Journal with Person (rel.: 2 – 1)
14. Read Journal
15. Delete Journal
16. Read Journal
17. Delete Person / should fail
18. Read Person
19. Delete Journal
20. Read Journal
21. Delete Person
22. Read Journal

Note that also, in this case we preserve at the end of the test case execution the original state of the database. On the other hand, this criterion subsumes the previous with one table, because the states of the table Person are part of the states of this pattern, and all the transitions of the first example are also included in this one. That means that if we find and generate test cases for a two tables' relation, it is not necessary to worry about generating test cases for each table apart.

C. *Three-table Patterns*

In the previous subsection, we are not including a type of binary relation at a conceptual level, which are the *many to many* relations, because at a database level it is implemented with three tables: two tables with the data of the entities, and another auxiliary table to store the relations, referencing the

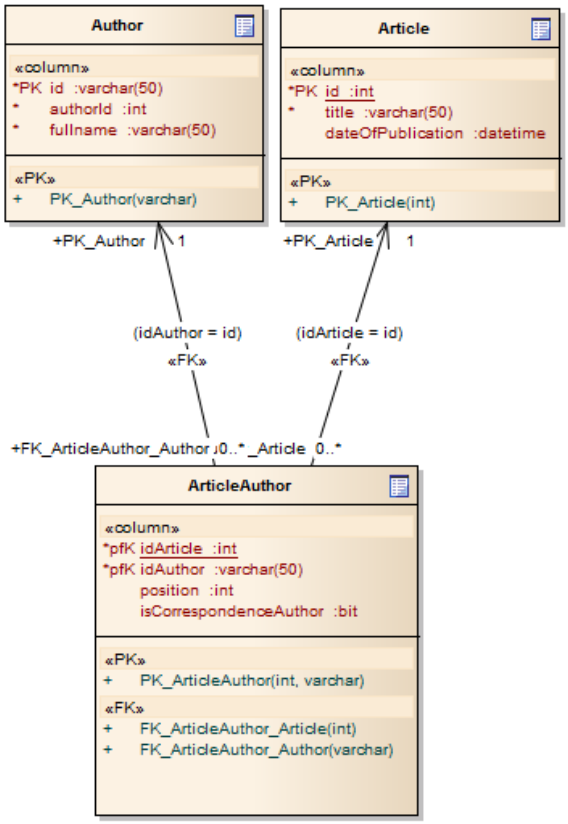


Figure 6 - Example with 3 tables

primary keys of the entities, defining its own primary key as the addition of the primary keys' attributes. Also, could be more attributes in the auxiliary table to store data related with the association. Paying attention to *Figure 6*, we can see the relation between *Authors* and *Articles*, where the same author can have many articles, and the same article can have many authors. The relation table *ArticleAuthor* has some attributes to store information about the relation between the *Article* and *Author*, for example indicating if this this author is corresponding author for this article.

This particular case imposes some considerations. We should add some valid states to the previous state machine, including association ends: 2 – 1 and 1 – 2, and 2 – 2. Also, the update of the relation table has two foreign keys, therefore there will be two special *update* operations that have to consider to reference valid (existent) and invalid (inexistent) tuples in the referenced tables.

In this example is interesting to mark something else that is that the tester has the possibility to add extra information to the data model, in order to validate some aspects of the logic that cannot be represented in the database schema. In the relation between *Article* and *Author*, perhaps it does not make sense to have an article without any authors, but this cannot be implemented in the schema, it must be managed in the logic, therefore, we want to check it. So, after the reverse engineering process we could modify the data model in order to generate test cases that can verify this kind of situations, just changing the association end multiplicity from "0 – *" to "1 – *".

IV. Related Work

Regarding test data generation for systems with databases, Tuya et al. [12] define a coverage criteria based on SQL queries, applying a criteria similar to *Modified Condition/Decision Coverage* [13] but considering the conditions of FROM, WHERE and JOIN sentences, generating test data to cover this criterion. There are some approaches (from Haller et al. [14] and Emmi et al. [15]) where the code coverage criteria are extended in order to consider the embedded SQL sentences, generating database instances to cover the different scenarios proposed as interesting. Arasu et al. [16] propose to specify in some way the expected results of each SQL included in the test, and then they can generate test data to satisfy this specification. The proposal from Chays et al. [17], called AGENDA, takes as input the database schema and categorized test data given by the user, whereby generates test cases and initial database states, and validating after the test case execution the outputs and the final database state. Neufeld et al. [18] generate database states according to the integrity restrictions of the relational schema, using a constraint solver. As far as we know, many proposals for test data generation exist, but none of them focuses on automated test model generation using model transformations.

There are various proposals to generate test cases automatically from UML models, as the ones described by Offut et al. [19] and Brucker et al. [20], but as far as we known, only Fujiwara et al. [21] proposed a special consideration for information systems with databases. In this work, they propose to generate test cases considering a UML class diagram to represent the data model, and another to represent the screens. The data restrictions (foreign keys, relations between data inputs and database fields, etc.) and pre and post conditions of the methods under test are represented with *Object Constraint Language* (the OMG's standard rules definition language). The whole test model must be specified manually, and therefore, maintained. The test cases generated are centered on the given restrictions, while in our proposal we pay attention on the data model automatically obtained, without maintenance costs.

V. Conclusion and Future Work

This paper has presented a method for test case design based on the data model, what is useful for our framework to test information systems with databases. From a well-designed database we can validate, with few extra effort, that the logic that manages the structures does it correctly.

This approach could be applied for any kind of system that uses a data base. We are developing the first group of patterns in order to put it into practice and validate our ideas, and to compare with other approaches. We believe that we can save time and effort detecting many errors before to deliberate a version to the testing team. Doing so, we can let a tester concentrate in the hard and more interesting task of testing the complex business rules of a system.

Another important point within the future work is related with complex objects types for the columns, as well as complex rules taken from checks or from the source code.

We also want to validate the scalability of the idea. For each entity it is necessary to implement some adaptation layer, but then the test cases executes completely automatically, independently of the amount of patterns defined.

Moreover, as a future work, we plan to experiment with different kind of model-driven development tools, as GeneXus [22] or OOH4RIA [23], because this kind of tools generate the system code from data models in a structured way, what could permit us to generate automatically the adaptation layer, in order to generate executable test cases with no extra cost.

Acknowledgment

This work has been partially funded by the *Agencia Nacional de Investigación e Innovación (ANII, Uruguay)*, by DIMITRI project *(Desarrollo e Implantación de Metodologías y Tecnologías de Testing, TRA2009_0131, Spain)* and by MAGO/Pegaso project *(Mejora Avanzada de Procesos Software Globales, TIN2009-13718-C0201, Spain).*

References

[1] P. Baker, Z.R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*. 2007: Springer-Verlag New York, Inc.

[2] OMG. *Unified Modeling Language*. 1997 [retrieved: october, 2012]; Available from: http://www.uml.org/.

[3] D. Gornik, *UML Data Modeling Profile*. 2002, IBM, Rational Software.

[4] S. Yin and I. Ray. *Relational database operations modeling with UML* in *AINA'05: Advanced Information Networking and Applications*. 2005. Vol. **1** pp. 927-932.

[5] G. Sparks, *Database modeling in UML*, in *Methods & Tools*. 2001. pp. 10-22.

[6] K. Zieliriski and T. Szmuc, *Data modeling with UML 2.0*. Software engineering: evolution and emerging technologies, 2006. Vol. **130**: pp. 63.

[7] F. Toledo, B.P. Lamancha, and M.P. Usaola. *Towards a Framework for Information System Testing - A model-driven testing approach* in *ICSOFT*. 2012. Rome, Italy.

[8] A. Andrews, R. France, S. Ghosh, and G. Craig, *Test adequacy criteria for UML design models.* Software Testing, Verification and Reliability, 2003. Vol. **13** (2): pp. 95-127.

[9] M.P. Usaola and B.P. Lamancha. *CTWeb*. [retrieved: october, 2012]; Available from: http://alarcosj.esi.uclm.es/CombTestWeb/.

[10] T. Koomen, L. van der Aalst, B. Broekman, and M. Vroon, *TMap Next, for result-driven testing*. 2006: UTN Publishers.

[11] M. Fewster and D. Graham, *Software test automation: effective use of test execution tools*. 1999: ACM Press/Addison-Wesley Publishing Co.

[12] J. Tuya, M.J. Suárez-Cabal, and C. De La Riva, *Full predicate coverage for testing SQL database queries.* Software Testing Verification and Reliability, 2010. Vol. **20** (3): pp. 237-288.

[13] J.J. Chilenski and S.P. Miller, *Applicability of modified condition/decision coverage to software testing.* Software Engineering Journal, 1994. Vol. **9** (5): pp. 193-200.

[14] K. Haller. *White-box testing for database-driven applications: A requirements analysis*. 2009: ACM, pp. 13.

[15] M. Emmi, R. Majumdar, and K. Sen. *Dynamic test input generation for database applications* in *ISSTA'07: Software Testing and Analysis*. 2007, pp. 151-162.

[16] A. Arasu, R. Kaushik, and J. Li. *Data generation using declarative constraints* in *International conference on Management of data*. 2011: ACM, pp. 685-696.

[17] D. Chays and Y. Deng. *Demonstration of AGENDA tool set for testing relational database applications*. 2003: IEEE Computer Society, pp. 802-803.

[18] A. Neufeld, G. Moerkotte, and P.C. Loekemann, *Generating consistent test data: Restricting the search space by a generator formula*. The VLDB Journal, 1993. Vol. **2** (2): pp. 173-213.

[19] J. Offutt and A. Abdurazik, *Generating tests from UML specifications.* «UML»'99—The Unified Modeling Language, 1999: pp. 76-76.

[20] A. Brucker, M. Krieger, D. Longuet, and B. Wolff, *A specification-based test case generation method for UML/OCL*. Models in Software Engineering, 2011: pp. 334-348.

[21] S. Fujiwara, K. Munakata, Y. Maeda, A. Katayama, and T. Uehara, *Test data generation for web application using a UML class diagram with OCL constraints.* Innovations in Systems and Software Engineering, 2011: pp. 1-8.

[22] Artech. *GeneXus*. 1988 [retrieved: october, 2012]; Available from: http://www.genexus.com

[23] S. Meliá, J. Gómez, S. Pérez, and O. Díaz. *A model-driven development for GWT-based Rich Internet Applications with OOH4RIA*. 2008: Ieee, pp. 13-23.

 ISBN: 978-1-61208-233-2

A Model-Based Approach to Validate Configurations at Runtime

Ludi Akue, Emmanuel Lavinal, Michelle Sibilla
IRIT, Université de Toulouse
118 route de Narbonne
F31062 Toulouse, France
Email: {akue, lavinal, sibilla}@irit.fr

***Abstract*—Dynamic reconfiguration is viewed as a promising solution for today's large scale and heterogeneous computing environments. However, considering the critical missions networked systems support, dynamic reconfiguration cannot be achieved unless the accuracy of its behaviors is guaranteed. For that reason, dynamic reconfiguration solutions should provide validation capabilities to ensure the correctness and the safety of reconfiguration activities. Current solutions mainly address use-case specific configuration validation or fail to handle the additional operational validity requirements induced by dynamic reconfiguration. In this paper, we describe a model-based approach for validating configuration changes at runtime. The approach is based on MeCSV, a metamodel that allows a platform and vendor-independent specification of a reference model, that is, the configuration schema of the managed system as well as constraints that should be respected for structural consistency and operational compliance. We provide an overview of the MeCSV language and demonstrate the feasibility of this approach using a messaging platform case study.**

Keywords-dynamic reconfiguration; configuration validation; configuration specification; model-based approach.

I. INTRODUCTION

The self-management vision has gained a lot of momentum in networked systems management where it is viewed as a promising solution for today's large scale and heterogeneous computing environments management. This vision consists mainly in endowing managed systems with self-adaptation capabilities to maximize their usability [1].

Regardless of the management functional domains (e.g., fault, performance, security), dynamic reconfiguration activities are the principal means through which self-management is carried out. However, dynamic reconfiguration capabilities should not endanger the system's operation, otherwise they would nullify the expected benefits: reconfiguration validation is one of the fundamental issues that conditions dynamic reconfiguration effectiveness [2]. Consequently, management systems should support online validation to guarantee the correctness and the safety of reconfiguration activities.

This paper complements previous work on defining a framework for dynamic reconfiguration validation. In [3], we argued that runtime reconfiguration validation should go beyond traditional structural sanity checks to further assess the safety of candidate configurations regarding operational conditions at hand. For example, when a max request size is erroneously set smaller than the current number of requests sent to a process, it can introduce some inconsistencies thus compromise the system's operation. In other words, in the matter of self-configurable systems, prevailing operational states can invalidate the suitability of a runtime produced configuration no matter its structural correctness. Consequently dynamic reconfiguration validation should consider an *operational applicability validation* which consists of validating proposed configuration changes against the current system's operational state to test the suitability of its deployment.

In this paper, we present a model-based approach for configuration specification that enables a platform-independent validation of configuration modifications at runtime.

The approach is based on a metamodel we develop named MeCSV (Metamodel for Configuration Specification and Validation). MeCSV implements appropriate constructs that allow vendors or operators to define their own reference model that every valid configuration instance should conform to, independently from management platforms and configuration protocols in use.

Indeed, MeCSV provides an intermediate high-level language that resolves the heterogeneity of configuration information and semantics. It also includes rule specification features to define different types of constraints to be validated dynamically on specific configurations produced at runtime. Finally, MeCSV incorporates constructs to represent monitored data of interest that will serve to assess the operational compliance of a given configuration instance.

In particular, one novelty of the metamodel is to include the capability to express both offline and online constraints. The former allows operators to define structural integrity rules while the latter allows them to define rules to be enforced regarding operational conditions, necessary to ensure the operational validity of produced configurations.

The remainder of the paper is structured as follows: Section II presents related work and Section III includes a case study that will be used throughout the article to illustrate usage examples of the MeCSV metamodel. Section IV introduces the validation approach we propose, built upon the MeCSV metamodel whose core constructs are described in Section V. Finally, Section VI describes implementation

 ISBN: 978-1-61208-233-2

details of a prototype experiment and Section VII concludes the paper and identifies future work.

II. Related Work

The need for configuration representation standards and configuration automation are growing concerns regarding the complexity of the configuration management of today's large-scale and heterogeneous systems [4], [5]. Our work is at the junction of these two topics as the MeCSV metamodel enables a generic and vendor-independent configuration specification and runtime validation which is a prerequisite for configuration automation as well as self-configuration.

Most related work proposes platform-dependent data models that principally provide structural integrity checks of functional configuration parameters [2], [6], [7], [8] and consider to a lower extent the validation of non-functional configuration parameters whose values depend on ongoing operational conditions (e.g., QoS, resources utilization). The novelty of our approach is to provide a language that is designed specifically for dynamic validation, it addresses both structural and operational validity.

The DMTF Common Information Model (CIM) [9] and the YANG data modeling language [10] include constructions to model configuration data. CIM provides particularly *SettingData* and the *OCL qualifier* constructs that can be used to indicate configurations and constraints to be respected, however, these elements are close to manual configuration, thus not flexible for a runtime reconfiguration environment. YANG provides a flexible data modeling language with means to specify structural constraints that will be enforced at runtime. However, YANG is specific to the Network Configuration Protocol (NETCONF) [11].

Our work also relates to PoDIM, a high-level language that allows to describe configurations as well as express the structural constraints that should be respected during managed objects creation and modification [7]. Even though they also define a high-level language for configuration specification, the two approaches are different since PoDIM is used to generate valid configurations (from rules defined by an administrator) whereas we validate configurations produced by existing management systems. In contrast to PoDIM, we also addresses the operational compliance issue.

Configuration validation is also addressed as a Constraint Satisfaction Problem [12], [13]. Nevertheless, the considered constraints are structural and static and their satisfaction does not consider the operational environment that can condition the applicability of generated configurations. A runtime validation is still required to assert the operational compliance of generated configurations regarding runtime conditions variations.

III. Use Case

This section introduces a Message-oriented Middleware (MOM) use case on which the examples given throughout the following sections will be based.

MOM systems are profitable to integrate heterogeneous and distributed applications seamlessly by making use of messaging servers to mediate communications between them. One other advantage is that by adding a management interface, an operator can monitor and manage the system's performance, reliability and scalability without losing function. Validating a MOM system's runtime evolving configurations is a suitable scenario for the evaluation of the approach we propose. The formalisms we will rely on respect the JORAM MOM configuration description [14].

A JORAM platform provides the following configurable features: message servers that route and deliver messages, destinations that are physical storages supporting either a point to point messaging (queue) or a "publish/suscribe" messaging (topic), connection factories used to enable client connections to the message servers according to used connection protocols (e.g., TCP).

Figure 1 presents the distributed JORAM platform configuration example that will be used in Section VI (reconfigurations scenarios). It consists of three servers S0, S1, S2 respectively providing queue-type destination (Qa, Qb, Qc, Qd and Qe) and TCP connection services to client applications.

Configuring this example platform consists in configuring each server, that is setting servers' local parameters (e.g., identifier, name, hostname) and the configuration parameters of the hosted elements (services, connection factories and destinations).

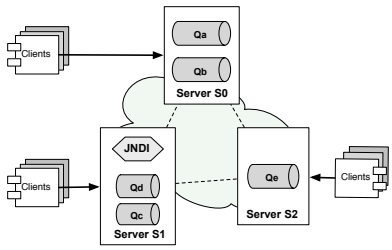


Figure 1. Use case system architecture

The following requirements are considered for the purpose of the case study:

- Configuration structure: It should respect the platform's architecture and the relationships between the configuration parameters. (`Req1`)
- Naming service: Connection factories and destinations should be accessible via a naming service i.e., the platform should provide an accessible JNDI service where the administered objects should be stored. (`Req2`)
- Memory optimization: The queue memory should not run low in memory, i.e., the queue should not be loaded at more than 80% of its maximum capacity. (`Req3`)

 ISBN: 978-1-61208-233-2

IV. CONFIGURATION VALIDATION APPROACH

The goal of our work is to provide means to enable an automatic configuration validation in self-configurable systems. Concretely, we want to build a validation system capable of automatically asserting the correctness and safety of configuration data at runtime, that is checking that configuration values remain within authorized bounds and do not compromise intended service behavior. To meet this objective, we follow a model-based approach in which we define a lightweight, yet consistent metamodel that provides constructs for a vendor neutral configuration data description and a constraint-based validity enforcement.

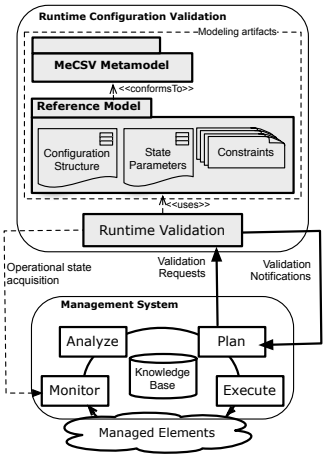


Figure 2. Proposed model-based configuration validation approach

The aim of this metamodel is first, to allow operators to specify their system's configurations thanks to appropriate constructs and rules; second, to enable the automatic validation of runtime proposed configurations against this model independently of both management platforms and configuration protocols.

As depicted in the upper part of Figure 2, the metamodel we propose is used to specify a *Reference Model* that every possible configuration of the target system should conform to. This reference model includes the *configuration's structure* (configuration parameters) as well as the different *constraints* every valid configuration should respect. The novel aspect of these constraints is to cover both structural integrity and operational applicability validation:

- Structural integrity validation checks the correct structure and composition of configuration parameters in terms of authorized values and consistent cross-components dependencies. For example, checking that a *host-address* configuration parameter exists and is well formed according to the IPv4 or IPv6 format.
- Operational applicability validation checks if the configuration fulfills the runtime operational conditions. For instance, assessing that `Req3` still holds after a configuration modification. This type of validation requires the knowledge of the current runtime context. The reference model thus includes the concept of *state parameters* for the acquisition of necessary monitored data.

Note that the reference model is to be defined by the human operator according to system and management requirements. Then, it will be used at each dynamic reconfiguration decision to verify produced configuration instances.

The reference model can also be modified, for example with the addition, removal or modification of constraints or configuration elements at any time during the management system's life cycle if needed.

The process for validating proposed configurations at runtime will work as follows: the reconfiguration decision function of the management system (the *Plan* block in the lower part of Figure 2) will interact with the runtime configuration validation. Every produced configuration instance will be dynamically checked against the reference model and be consequently validated structurally and operationally before deployment.

V. MECSV OVERVIEW

This section presents the salient features of the metamodel depicted in Figure. 3. MeCSV has been formally specified as a UML profile [15] to ease the usage of the MeCSV language and benefit from the abundance of UML modelers.

A. Configuration Data Description

Configuration data are generally described in some configuration files where their structure is specified through the setting of some configuration properties with appropriate values and options. Additionnally, bindings between system's elements need to be reflected in their configurations, for example, the coordination of the server's hostname value with the machine's hostname value. This part of the metamodel represents subsequents concepts to do so.

1) Configuration Parameter: represents quantifiable configuration parameters of managed elements; their expression defines the configuration data structure. For example, a message server's identifier or hostname information.

2) Configuration: acts as a container for configuration parameters allowing to coordinate them and to group them in categories. For example, a configuration file can be modeled as a single *Configuration*, or for more flexibility, divided into multiple *Configurations*.

3) Configuration Dependency: represents bindings between two configuration elements meaning a configuration parameter of one configuration references a whole or a part of the other configuration. Typically, a server's hostname references its host machine's name information.

 ISBN: 978-1-61208-233-2

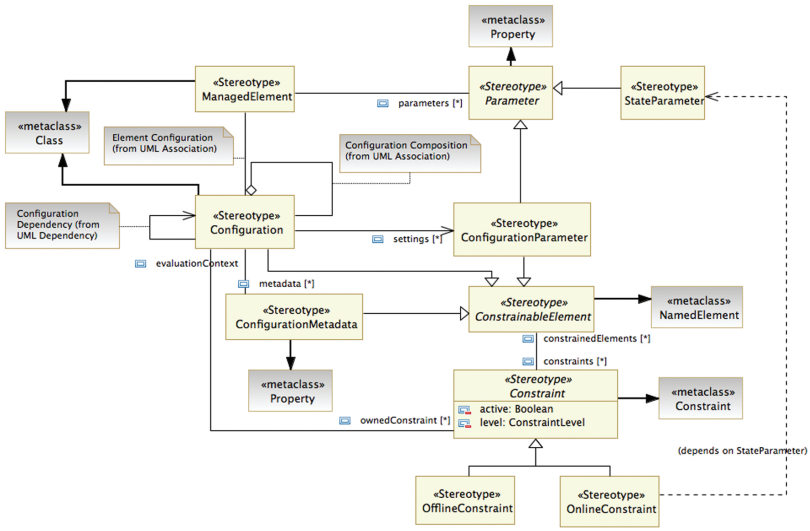


Figure 3. Core of the UML profile for the MeCSV metamodel

4) Configuration Composition: allows to divide a main configuration into partial configurations. For example, a message server's configuration is split into message services, connection factories and destinations sub-configurations. It means that the complete server's configuration is the collection of its local configuration parameters and its associated sub-configurations.

5) Configuration Metadata: allows to specify metadata for configuration lifecycle management. For instance, one could want to tag specific configurations as default or initial. Another example is the `visited` metadata used in the JORAM platform to mark deployed configurations.

B. Connection to the Monitoring Framework

As our work targets a global management environment where the managed system is both observable and reconfigurable, we provide constructs to represent information about managed elements as well as their monitored state. A knowledge of the monitored state is required to guide reconfigurations and to assert the operational compliance of proposed configurations.

1) Managed element: represents the notion of managed element commonly defined in several management information models. A common pattern is to separate managed elements representation from configuration modeling, managed elements containing monitoring-oriented information.

2) State Parameter: models the traditional operational state attributes like operational status, statistical data, in sum, any monitored information. Enabling the access to their values is required to process online constraints. The number of pending messages or current active TCP connections are examples of state parameters.

In our approach, *Managed Element* and *State Parameter* are the necessary management building blocks for configurations and runtime constraints definition. Their values are supposed to be provided by an existing monitoring framework. They are considered as read-only elements.

C. Configuration Validity Enforcement

Defining a configuration data structure does not suffice to guarantee the validity of formulated configuration instances; the following elements allow to define the constraints that configuration instances should respect.

1) Constraint: represents the restrictions that must be satisfied by a correct specification of configurations according to the system's architecture and management strategies. `Req1`, `Req2` and `Req3` are examples of high-level level constraints limiting the range of allowable configuration parameters values. They will be translated into low-level constraints that can be enforced at runtime.

The *Constraint* element is subtyped into *offline* and *online constraints* to support the specificities of the two types of configuration validation.

2) Offline Constraint: represents structural integrity, that is rules for architectural compliance. They can be checked either beforehand at design time or during runtime and do not involve any check against monitored data. The following OCL expressions are examples of offline constraints derived from `Req1`: `self.jndiName<>null`, `serverId->include(parent.serverId)= true`. The first expression ensures that a queue has a registered name and the second guarantees that a queue is associated with a valid server.

3) Online Constraint: defines rules for the operational applicability enforcement. *Online constraints* use *state parameters* values to assess the operational compliance of configuration data. They are necessarily checked at runtime. `self.nbMaxMsg`$>80\% *$ `self.arrivalsCounter`,

`JNDIServer->include(operationalStatus= ON)` are examples of online constraints expressed in OCL. The former is a translation of `Req3`, the latter is derived from `Req2` and ensures that the configuration of the system includes a running JNDI service. They can only be evaluated against the current value of a queue's message load and the operational status of the naming service respectively.

Constraints also have a "constraint level" attribute to modulate their strictness together with an "active" attribute to activate or deactivate them depending on the operational context and management strategies (e.g., critical vs non-critical).

D. Usage Example

Figure 4 illustrates an application of the MeCSV UML profile to the modeling of a message queue according to specified `Req1`, `Req2` and `Req3` in Section III.

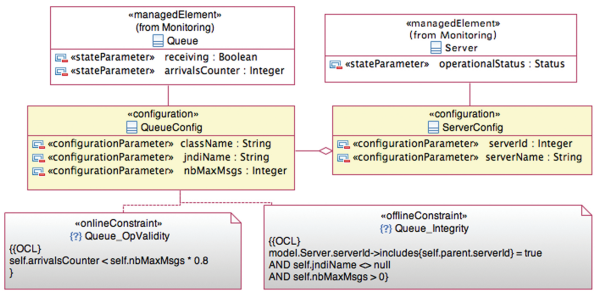


Figure 4. Excerpt of the reference model for a message queue

This reference model contains the configuration structure of a message queue, the offline and online constraints that should be respected and depending state parameters.

VI. Experiment

This section presents a prototype implementation of the approach applied to the MOM system configuration in Section III. The underlying objective is to evaluate the ability of MeCSV to serve as a formal specification notation, namely whether a MeCSV reference model can suffice to enable a runtime configuration validation.

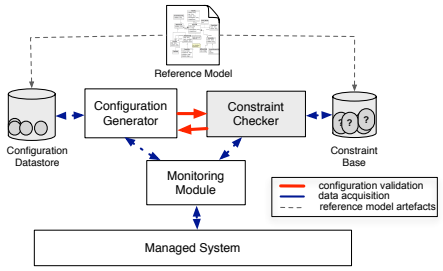


Figure 5. Architecture of a prototype implementation

A. Methodology

The prototype is constituted of three components that interact automatically as shown in Figure 5:

1) Architecture:

- A configuration generator: it outputs configuration instances according to defined reconfiguration scenarios.
- A monitoring module: it updates operational state metrics according to given monitoring scenarios.
- A constraint checker: it checks related configuration elements against the reference model. This constraint checker is specifically designed to interpret MeCSV constructs. It can thus process any given configuration data defined with the MeCSV language.

2) Implementation Details: The prototype was developed in Java:

- Each MOM system's element (i.e., servers, destinations,...) has two corresponding Java class representations for its monitoring and its configuration view. For instance, a message server is implemented through a *Server* class containing its state attributes and a *ServerConfig* class for its configuration attributes.
- The code of the configuration view is generated from the defined reference model thanks to MeCSV UML profile.
- Constraints are implemented as test functions. Their evaluations consist in appropriate method calls on related constrained elements.

3) Scenarios:

Reconfigurations scenarios: they covered typical performance tuning activities: the addition and removal of servers, the platform is scaled up and down (from a centralized configuration of a single server to a distributed one made of three servers: Figure 1) and the modification of queues's configuration parameters to adjust the memory usage, especially the variation of its maximum capacity.

Common structural flaws (missing mandatory values, omitted dependencies) are introduced programmatically into generated configurations to test the constraint-checking.

Monitoring scenarios: they covered operational statuses variations as generally observed in case of service failure or communication lost as well as performance decrease through message load variations than can possibly impact the platform's memory usage.

4) Execution: The runtime configuration generator periodically produces a new configuration instance and sends it to the constraint checker for validation while the monitoring module arbitrarily updates operational state values according to monitoring scenarios. The constraint checker evaluates input configurations by calling appropriate test functions. The constraint checker returns an OK message (no found errors) or a list of violation errors.

 ISBN: 978-1-61208-233-2

B. Discussion

Thanks to the metamodel, we described a MeCSV reference model of the use case system that comprised the system configuration schema, state data of interest and offline and online constraints that should be respected. A provided configuration generator produced configuration instances that were evaluated by a prototype constraint-checker. Since those configuration data are expressed using the MeCSV language, the constraint-checker seamlessly processed them and tested them against the available set of constraints. Violations were detected and reported during the execution of the different scenarios.

This preliminary experiment shows that the approach we propose is feasible. As long as there is a defined MeCSV reference model of the managed system, and that its runtime candidate configurations as well as its monitored data can be exported using the MeCSV format, our constraint-checker can be plugged in the related management system and perform an automatic and platform-neutral configuration validation.

Yet several points remain to clarify before practical usage:

- The design of the constraint checker: we are currently studying runtime OCL formats and compilers [16], [17] and their performance on scalable architectures.
- The interpretation of violation errors: one issue is the expressiveness of violation errors in order to guide the re-formulation of a new candidate configuration. This aspect can be included in the definition of a protocol between the reconfiguration decision and the validator.

VII. Conclusion and Future Work

Dynamic reconfiguration is an important issue if we are to build large, complex and heterogenous systems with an acceptable level of reliability. However, dynamic reconfiguration decisions should be validated before their application in order to guarantee the system's accurate operation.

This paper presented a model-based approach that aims to enforce the validity of runtime configuration changes. We have shown that configuration validation at runtime goes beyond structural correction checks to further verify the operational consistency of configuration modifications.

We proposed a metamodel (MeCSV) that provides platform and vendor neutral constructs for the specification of a system's reference model that is the system's configuration schema including structural and runtime constraints that should be respected. A dedicated constraint-checker can then consume the defined reference model and automatically validate output configurations against it.

MeCSV has been implemented as a UML profile and a preliminary experiment validates the feasibility of its usage to enable online configuration validation.

Future work intend to carry on our experiments on common systems to consolidate the genericity of our approach. Moreover, we are working on a complete framework to support the metamodel with an adequate runtime constraint checker.

References

[1] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe, "An Automated Formal Approach to Managing Dynamic Reconfiguration," in *ASE'06: Inter. Conference on Automated Software Engineering*, 2006, pp. 37–46.

[3] L. Akue, E. Lavinal, and M. Sibilla, "Towards a Validation Framework for Dynamic Reconfiguration (short paper)," in *IEEE/IFIP International Conference on Network and Service Management (CNSM)*, 2010, pp. 314–317.

[4] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, ser. USITS'03, 2003, pp. 1–1.

[5] P. Anderson and E. Smith, "Configuration tools: working together," in *Proceedings of the 19th conference on Large Installation System Administration Conference*, 2005.

[6] A. V. Konstantinou, D. Florissi, and Y. Yemini, "Towards Self-Configuring Networks," in *DANCE'02: DARPA Active Networks Conference and Exposition*, 2002.

[7] T. Delaet and W. Joosen, "PoDIM: A Language for High-Level Configuration Management," in *LISA*, 2007, pp. 261–273.

[8] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The SmartFrog Configuration Management Framework," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 16–25, 2009.

[9] "CIM Schema version 2.29.1 - CIM Core," june 2011.

[10] M. Bjorklund, "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)," Internet Engineering Task Force (IETF), RFC 6020, october 2010.

[11] R. Enns, "NETCONF Configuration Protocol," Internet Engineering Task Force (IETF), RFC 6241, december 2006.

[12] T. Hinrichs, N. Love, C. J. Petrie, L. Ramshaw, A. Sahai, and S. Singhal, "Using Object-Oriented Constraint Satisfaction for Automated Configuration Generation," in *DSOM*, 2004, pp. 159–170.

[13] L. Ramshaw, A. Sahai, J. Saxe, and S. Singhal, "Cauldron: a policy-based design tool," in *Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop on*, 2006, pp. 113–122.

[14] "Java (TM) Open Reliable Asynchronous Messaging website," september 2011. [Online]. Available: http://joram.ow2.org/

[15] "OMG Unified Modeling Language (OMG UML), Superstructure V2.1.2," november 2007.

[16] M. Gogolla, M. Kuhlmann, and F. Büttner, "A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, 2008, pp. 446–459.

[17] C. Avila, A. Sarcar, Y. Cheon, and C. Yeep, "Runtime Constraint Checking Approaches for OCL, A Critical Comparison," in *SEKE*, 2010, pp. 393–398.

 ISBN: 978-1-61208-233-2

Applying an MBT Toolchain to Automotive Embedded Systems: Case Study Reports

Fabrice Ambert*, Fabrice Bouquet*†, Jonathan Lasalle*, Bruno Legeard† and Fabien Peureux*†
*FEMTO-ST Institute / DISC department - UMR CNRS 6174
16, route de Gray, 25030 Besancon, France.
{fambert, fbouquet, jlasalle, fpeureux}@femto-st.fr
†Smartesting R&D Center
18, rue Alain Savary, 25000 Besancon, France.
{bouquet, legeard, peureux}@smartesting.com

***Abstract*—This paper illustrates the use of a Model-Based Testing approach from SysML test model using four complementary automotive case studies. The purpose of these experiments is to give an empirical evidence of the reliability and to show the suitability of this tooled approach for the validation of embedded mechatronic systems (systems mixing software and hardware aspects). The experimented toolchain, based on the Model-Based Testing principles, reuses well-known and effective existing tools in order to obtain an end-to-end toolchain from the modeling step to the execution of the concrete test cases derived from the initial test model. This fully automated toolchain and the four automotive case studies are introduced, and automation feedback is discussed.**

Keywords*-*Model-Based Testing; Automotive Embedded Systems; Case Study Report.

I. INTRODUCTION

The growing complexity and intensive use of software embedded systems, combined with constant quality and time-to-market constraints, entail the implementation of high-performance and effective system validation strategies. Since functional testing is a strategic activity for software quality assurance, it creates new challenges for engineering practices in this domain. To address this activity, we propose to apply Model-Based Testing (MBT) approach to complete the manual test cases executed during the software integration, which often relies on manual, repeated and tedious efforts.

During the last decade, Model-Based System Engineering (MBSE) methodologies have emerged on the sharing and standardisation of embedded software technologies [1]. These approaches put a strong emphasis on the use of models at different steps of the system specification to increase the quality of the software design process. In this context, testing against original expectations can be done using Model-Based Testing approach [2]. MBT is a particular type of software testing techniques in which test cases are automatically derived from a high-level model, which describes the expected behavior of the System Under Test (SUT). MBT is an increasingly used approach that has gained much interest in recent years. Today, it is getting closer and closer to an industrial reality: theoretical concepts (and associated tools) to derive test cases from specifications are indeed now mature enough to be applied in many application areas [3]. However, MBT approaches have still to provide a better degree of automation, especially to translate the generated test cases into executable test scripts in order to shorten the testing time and increase the global time-to-market [4].

The global picture of an MBT process is shown in Figure 1. The first step of this approach consists in specifying a model that captures the functional behavior of the SUT. From this specification, a tool automatically generates test cases, which can be seen as an abstract execution trace of the system. These test cases are abstract because they are defined at the same abstraction level as the model representing the SUT. Afterwards, from the abstract test cases, a concretization step allows to produce, test scripts that can be directly executed either on a simulation platform of the system, or directly on the concrete system. The automation of such test generation process is a strategic issue, since it can replace the (so current) manual development of test cases, which is known as costly and error-prone [5].

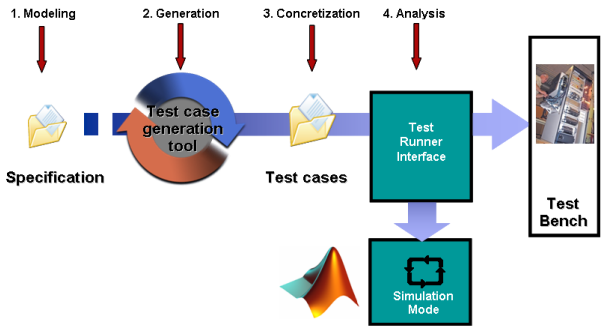


Figure 1. Model-Based Testing process.

In this paper, we illustrate the use of an MBT toolchain, providing an automated and repeatable process, dedicated to embedded and mechatronic systems, including real-time and continuous executions. We discuss the results using concrete case studies in order to show the effectiveness and the suitability of this end-to-end tooled MBT solution, but the relevance of the test cases is not discussed in this paper.

 ISBN: 978-1-61208-233-2

This paper is organized as follows. Section 2 presents an overview of the MBT toolchain, and defines each step of the test generation and execution process. Section 3 introduces four case-studies, conducted to evaluate the reliability of our tooled approach. Section 4 synthesizes our experience and gives feedback about automation issues. Finally, Section 5 gives conclusions and outlines our future work.

II. DESCRIPTION OF THE TOOLCHAIN

In this section, we briefly describe the toolchain implementing the MBT approach. This toolchain has been initially developed during the French project VETESS (from September 2008 to August 2010) and experimented during the last three years. The resulting MBT toolchain is based on the Smartesting MBT process [6], which has been adapted to address the specific testing needs and requirements of the automotive domain. To achieve this goal, it takes, as input, test models specified using the SysML [7] language, from which specific model coverage criteria have been created to generate dedicated test cases for embedded system validation. Concerning technical issues, we have developed a toolchain providing a full automated MBT solution from the test model to the execution of the generated test cases on the targeted SUT. This toolchain has been achieved by using the open-source and Eclipse-based modeling tool *Topcased*, the test generation engine *Smartesting Test Designer*^{TM}, and the test manager and execution environment (dedicated to embedded systems *Clemessy TestInView* platform). To ensure a fully automated process, interfaces between these tools have been developed. Before introducing the overall toolchain, each tool is now briefly described in the next subsections. A more detailed presentation of this toolchain is available in [8].

A. SysML modeling with Topcased

UML is widely used as a modelling support in industrial context and is today the main specification language for object modelling. Recently, to provide sufficient features to make it useful for systems engineers, SysML profile has been created. Even if SysML is a recent modeling language, it is on the rise in the industrial domain to specifically address system engineering issues. Thus, several modeling tools already support SysML models, such as Topcased, which means Toolkit in OPen-source for Critical Application and SystEms Development. We have decided to use this tool because it provides a SysML editor based on the UML metamodel (and therefore compliant with the OMG UML standard and the SysML metamodel, derived from the OMG SysML Profile).

More precisely, the test model is specified on the basis of a subpart of SysML notation called SysML4MBT [9]. A SysML4MBT model contains at least one Block Definition Diagram to represent the static view of the system (with blocks, associations, compositions, enumerations, properties, operations, signals, flow ports, etc.), at least one Internal Block Diagram to formalize interconnections between blocks, and at least one Statemachine diagram to specify the dynamic view of the system. In addition, Object Constraint Language (OCL) [10] expressions are associated to the SysML block operations and state diagram transitions to provide the expected level of formalization and precisely describe the dynamical behaviors of the system. Indeed, OCL is an unambiguous language that allows formally to express essential behavioral aspects of the SUT. That is why the combination of OCL and the object-oriented graphical model is known as a good practice to model the exact service the system has to do.

B. Test generation with Smartesting Test Designer^{TM}

Smartesting company has released an Eclipse-based tooled MBT solution to generate and manage functional tests from behavioral models specified in UML/SysML. Basically, automatic test generation algorithm carries out a systematic coverage of all behaviors of the test model by applying *All-Transitions* criterion. Moreover, to address the specificities of embedded systems, tests also cover each couple of receipt/sending signals: for each sending event and each corresponding receipt event, the coverage of the succession of the sending event and the receipt event is guaranteed.

Each test corresponds to a sequence of operations (or events) taking the form of a 3-part structure: a first subsequence places the system in a specific context (preamble) to exercise the test goal, a second subsequence invokes the behavior to be tested (test goal), and finally a last subsequence allows to return in the initial state so that test cases can be executed automatically in one single sequence. It should be noted that this 3-part structure can be completed by one or more observation function calls, which allow observing the system state at any time during the test execution (to make the verdict assignment more relevant). Indeed, the precise meaning of SysML4MBT permits to simulate the execution of the model, and thus use it as an oracle by predicting the expected output of the SUT.

The generated abstract test cases are finally exported into XML proprietary files from which the generated test cases are translated into specific languages or environments.

C. Test execution with Clemessy TestInView

TestInView (TIV) [11] is a test execution platform based on a National Instruments hardware architecture (NI TestStand) [12]. It is designed to generate and acquire simple or complex electric signals and to import mathematical models (as Matlab/Simulink) that simulate the behavior of an item of equipment that is absent from its future working environment. This platform can be used to describe the test sequences, execute them and automatically assess the expected results.

 ISBN: 978-1-61208-233-2

D. Overview of the toolchain

The built toolchain is depicted in Figure 2. The associated test process is defined as follows:

1) A SysML test model, specifying the SUT, is built using Topcased.
2) This SysML model is translated into a SysML4MBT model, which is exported to Test Designer^{TM}.
3) Test Designer^{TM} automatically generates abstract test cases from the model by applying coverage criteria, and produces the expected behavior of the SUT.
4) The generated test cases and expected outputs are then exported into TestInView platform. During this step, a manually-designed mapping table concretizes the abstract generated test cases into concrete scripts.
5) Finally, Clemessy TestInView platform allows to automate the test case execution on a simulated system or on a physical test bench. It also manages the verdict assignment by comparing automatically the execution results to the expected ones.

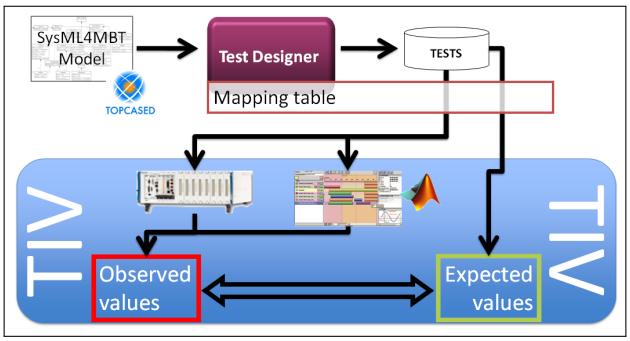


Figure 2. Overview of the MBT toolchain

The next section introduces the case studies used to illustrate how this MBT toolchain has been successfully applied to automotive embedded systems.

III. Case Studies

We now present four case-studies that have been used to experiment the MBT toolchain presented in the previous section. The goal of this work was to empirically show that such a tooled approach using SysML notation is suitable within the automotive embedded system context. The two first case-studies (front lightings and seat control system) can be seen as preliminary toy examples: they have been conducted to experiment only the modeling and the test generation process. The next two case-studies (front wiper and steering column) have been used to validate the entire toolchain from modeling to execution of the generated test cases (using either simulation framework or physical test bench). The functional scope of each case study is given. Metrics about the model structure are summarized in Table I on page 6. These data and the effort to conduct the case studies are discussed in Section 4.

A. Front Lightings

The first case study concerns the study of a car front lighting system. This system allows to put the dipped lights and full lights on and off. Unlike traditional lighting systems, we replaces the control stick by a tactile panel (also called control panel). This panel is composed of a dynamic screen (variable display) and a tactile surface. In the initial state, the panel and the lights are turned off. When the ignition is turned on, all lights stay turned off and the control panel is started. Two functionalities become then available: light on dipped lights or flash lights. Two different area are therefore displayed on the control panel screen. If we choose to light on dipped lights, other functionalities are ready for use: light on full lights, light off dipped lights or flash lights. If the user lights on full lights, dipped lights are automatically light off. From this new state, it is always possible to flash lights.

This case study proposes a system with quite simple communications (see Figure 3) but offers a quite complex statemachine by the number of possible fireable transitions.

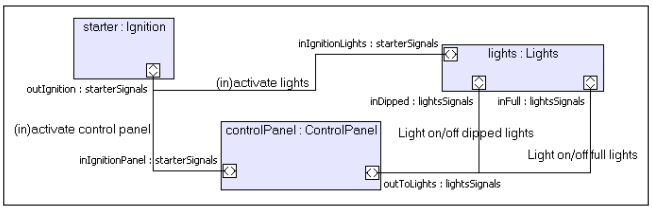


Figure 3. Internal Block Diagram of the front lightings case study

This model has generated 41 test targets that are covered by 11 abstract test cases. Since we did not have concrete test bench for this case study, it has been used to adjust our approach on modeling and test generation parts.

B. Seat control

The second case study was carried out on part of the electronic control of a car driver seat management (the specification of this case study is provided by [13]). As for the previous case study, neither a test bench, nor a simulator were available to execute generated tests. So, this case study has been also useful to validate the two first parts of the toolchain: modeling and test generation.

As shown in Figure 4, this system is composed of six motors (`LA`, `FH`, `RH`, `SD`, `B` and `HR`) that allow to change features of the seat. Each motor has a maximum amplitude. All motors can turn on in two different ways (`PLUS` and `MINUS`). They are divided in two groups: the first one containing `LA`, `FH` and `RH` motors, the other one containing `SD`, `B` and `HR` motors. We assume that, in a given time, only one motor can be running in a given group. Priorities are associated to each motor: if a higher priority motor is turned on during a lower priority motor is running, this second one is temporarily turned off in order to run the first one.

 ISBN: 978-1-61208-233-2

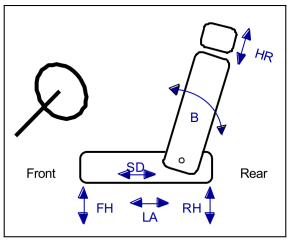


Figure 4. Seat control system

This example describes a continuous system: variations of the seat features are synchronized by a clock. The amplitude of motors is thus represented as an amount of clock ticks. The block definition diagram of this model is composed of one block for the buttons that activate motors (called `command`), one block for each motor and one block for the clock management. For instance, the statemachine of the `command` block is depicted in Figure 5.

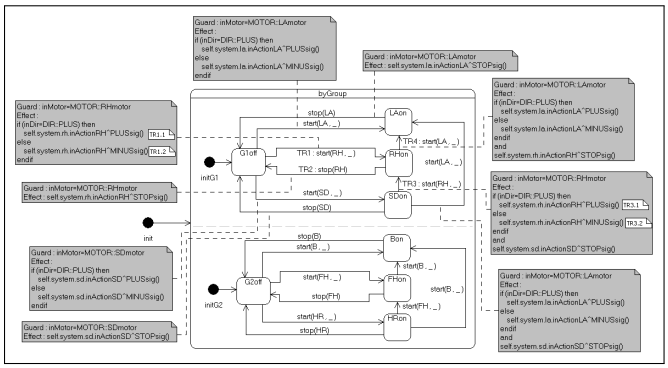


Figure 5. Statemachine of the command block of the seat control model

The global model contains 42 signals sendings and 48 signal receipts. The test generation strategy generated 130 test targets, which are covered by 78 abstract test cases.

C. Front Wiper

The third case study specifies a wiper system of a car. Modeled functionalities are drying up with different speeds (low, high and intermittently) and a window cleaning with drying up. In this system, a lot of mecatronic parts are considered: the serial link, the CAN bus and the EEPROM memory. Then, the model contains more transitions than previously (91 transitions shared by 12 parallel statemachine diagrams) and communications are much more complex. Thereby, 189 abstract test cases have been generated to cover the 233 targets derived from the SysML test model.

These generated test cases have been concretized and exported to the TestInView platform. As shown in Figure 6, tests have been executed on a simulation model (designed using Matlab). The result of the test execution on the simulator has been automatically compared to the expected result predicted by the SysML test model.

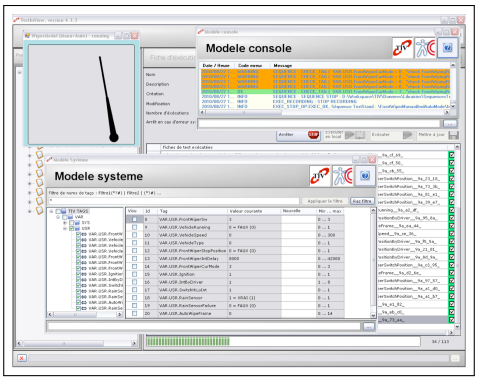


Figure 6. TIV simulation GUI of the front wiper system

D. Steering Column

The steering column case study aims to analyze the behaviors of a car steering column. A major issue of this last case study concerns the strong continuous feature of this system (its state is always evolving), which cannot be trivially abstracted. Indeed, variation of the steering column depends on complex mathematical formula and cannot be modelled using a SysML4MBT model, which describes only discrete actions. Because of these limitations, our approach consists in modelling the environment of the SUT in a discrete manner, and in deferring the management of continuous time issues at the concretization step. Thus, for this case study, statemachine diagrams are not used to represent behaviors of the SUT, but to represent behaviors of its environment. So, the road plots are modeled, and the expected values of the SUT are computed in a latter step by simulation (see Figure 7). Then, the testing process consists in comparing the values obtained using simulation against the values observed in the concrete system.

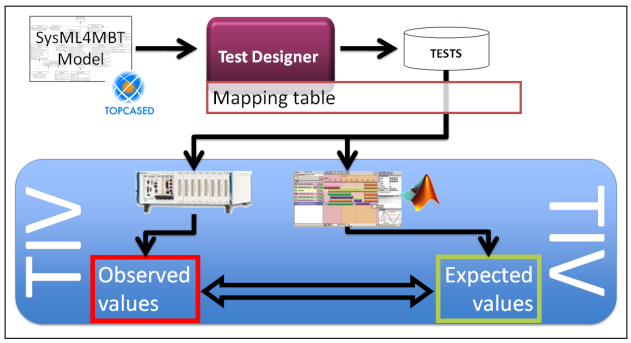


Figure 7. Tests execution for continuous systems

The SysML model represents road characteristics with blocks that are linked to the steering column (defining the black box SUT). Figure 8 depicts one of the 61 generated test cases. Perpendicular lines separate the different steps of the road. A flat road is represented by gray line, a downhill part by light gray line, an ascending part by black lines, and finally the various banking by arrows.

 ISBN: 978-1-61208-233-2

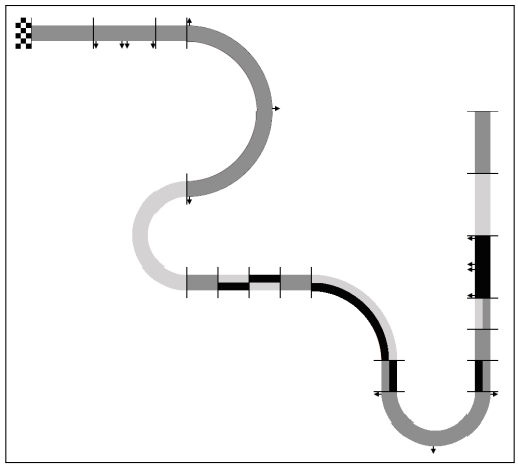


Figure 8. Graphical test generated for the steering column case study

Since the generated test cases do not allow to calculate the expected values (road plots do not give the status of the steering column), it is then necessary to execute the generated tests on a simulated Matlab version (Figure 9), and thus compare these results to the execution on the physical test bench (Figure 10). This comparison has been automated using the TIV framework. The execution of such scenario on this test bench is available at the end of the video in [14].

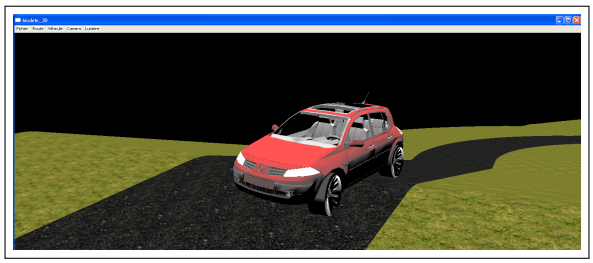


Figure 9. Simulator GUI of the steering case study

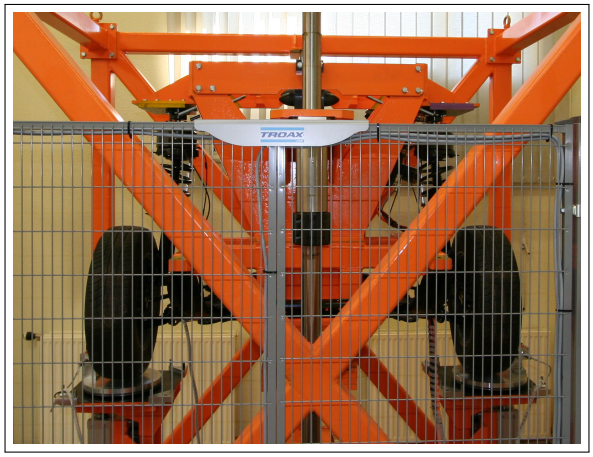


Figure 10. Physical test bench of the steering case study

IV. Experiment synthesis and Feedback

The four case studies, presenting a growing complexity in terms of model expressiveness and behavioral aspects (see Table I), have shown that Model-Based Testing from SysML can be successfully applied to several aspects of automotive embedded system domain. This tooled approach leads to great benefits to generate automatically test cases by ensuring a given model coverage and generating a very large number of test cases from a simple model. Moreover, for any change in the model, it offers the capacity to re-generate and re-execute the test cases automatically. As illustrated in Table I, the test generation time was always trivial in comparison to the time spent to write the model. Indeed, the complexity of the test models being reasonable, test generation tools, such as Test Designer^{TM}, are now mature enough to be efficient in terms of generation time and model coverage rate. However, with more complex and larger systems, a risk of combinatorial explosion during test case generation may occur.

In addition, it should be noted that our MBT process (that relies on a discrete representation of the SUT) can be nevertheless relevant even if the SUT refers to continuous issues (eg. steering column example) that cannot easily be abstracted (such as seat control example). In this specific context, the test model can be used to describe the dynamic of the SUT environment, meaning how the SUT can be stimulated by its environment (and not how it evolves against these stimuli). The expected behaviors of the SUT are computed latter, during the concretization step of the process, which then appears more complex than a simple mapping between abstract and concrete data.

Whatever the configuration may be, these experiments have shown that more than 50% of the time is consummed to manually design and manage the mapping table, which gives the relation between the concepts of the abstract test cases and the concrete sequences to be executed on the real system. The difficulty of this task often comes from the real-time features of the concrete system, and the need to synchronize all the operation calls of the test cases. The mapping between abstract and concrete notions has been clearly identified as the key point to make the automation of the concretization step manageable and reliable in an industrial context. This issue is not due to our technologies: previous works using other MBT tools have already underlined this rough step [15].

Finally, on the basis of this fully automated toolchain, new experiments are necessary to determine more precisely the scalability of our MBT approach. Moreover, real-life experiments with more complex and larger test models should be conducted to study in a deeper way the relevance of the generated test cases (our study was mainly focused on feasibility).

 ISBN: 978-1-61208-233-2

		Lightings	Seat control	Wiper	Steering
SysML model	Blocks	4	9	15	9
	Connectors / sends / receipts	8/14/10	24/42/48	26/58/65	10/25/20
	Statemachines	3	8	12	6
	States per statemachine	[2,5,5]	[8,1,3,3, 3,3,3,3]	[1,1,1,1,1,2, 17,10,2,2,2,2]	[2,4,3,6,3,4]
	Transitions per statemachine	[2,8,8]	[18,1,8,8, 8,8,8,8]	[2,3,2,4,1,3, 52,16,2,2,2,2]	[3,8,4,5,9,8]
Test results	Targets	41	130	233	106
	Tests	11	78	189	61
Effort	Modeling	99%	97%	40%	30%
	Test generation	1%	3%	4%	2%
	Concretization			50%	61%
	Test execution			6%	7%

Table I
SYNTHESIS OF EXPERIMENT RESULTS

V. CONCLUSION AND FUTURE WORK

This paper reported on results applied to the automotive system using an MBT toolchain prototype that automates the generation of executable test scripts from SysML test models. This prototype is based on existing tools that have been adapted and customized to achieve testing process automation: this prototype indeed offers an integrated approach and continuous process. Several case-studies have been successfully experimented and have showed that this toolchain is suitable and can gain benefits within automotive embedded system validation. However, the manual design and customization of the translation of the abstract test cases into concrete ones clearly appeared to be a pain. To provide a better degree of automation of this step, we intend to manage real-time issues at the earliest stage of the process, directly in the SysML model. To address this issue, we want to investigate the use of the UML MARTE profile [16]; this feature will allow to model and manage real-time constraints in the test model. In this way, the generated test cases will naturally consider the real-time requirements of the SUT, and thus will simplify the customization of the mapping table. Moreover, this extension will permit to define new test generation strategies, focusing on real-time issues.

REFERENCES

[1] J. Estefan, "Model-Based Systems Engineering (MBSE) Methodologies," MBSE Initiative and INCOSE Group, Survey INCOSE-TD-2007-003-01.B, June 2008.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing - A tools approach*, Morgan and Kaufmann, Eds. Elsevier Science, 2006, ISBN 0 12 372501 1.

[3] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, [retrieved: November, 2012]. [Online]. Available: http://dx.doi.org/10.1002/stvr.456

[4] A. Dias-Neto and G. Travassos, "A Picture from the Model-Based Testing Area: Concepts, Techniques, and Challenges," *Advances in Computers*, vol. 80, pp. 45–120, July 2010.

[5] H. Zhu and F. Belli, "Advancing test automation technology to meet the challenges of model-based software testing," *Journal of Information and Software Technology*, vol. 51, no. 11, pp. 1485–1486, 2009.

[6] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux, "A test generation solution to automate software testing," *Proceedings of the 3^{rd} Int. Workshop on Automation of Software Test (AST'08)*, pp. 45–48, May 2008.

[7] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2009, ISBN 9780123743794.

[8] J. Lasalle, F. Peureux, and F. Fondement, "Development of an automated MBT toolchain from UML/SysML models," *ISSE, Special issue of the Int. NASA Journal on Innovations in Systems and Software Engineering*, vol. 7, no. 4, pp. 247–256, September 2011.

[9] J. Lasalle, F. Bouquet, B. Legeard, and F. Peureux, "SysML to UML model transformation for test generation purpose," *Proceedings of the 3^{rd} Int. Workshop on UML and Formal Methods (UML&FM'10)*, November 2010.

[10] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1996, ISBN 0 201 37940 6.

[11] "Clemessy," http://en.clemessy.com/expertise/innovations/article/?tx_ttnews[tt_news]=9, [retrieved: November, 2012].

[12] "TestStand," http://www.ni.com/teststand/, [retrieved: September, 2012].

[13] M.-A. Peraldi-Frati, C. André, and J.-P. Rigault, "UML et le paradigme synchrone : Application à la conception de contrôleurs embarqués," *RTS'2002*, pp. 71–89, March 2002.

[14] "VETESS project," http://lifc.univ-fcomte.fr/vetess/, [retrieved: November, 2012].

[15] E. Dustin, T. Garrett, and B. Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Addison Wesley, 2009, ISBN 0321580516.

[16] O. M. Group, "UML Profile for MARTE, draft revised submission," OMG, OMG document number realtime/07-03-03L4.1, April 2007.