VALID 2014

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

ISBN: 978-1-61208-370-4

October 12 - 16, 2014

Nice, France

VALID 2014 Editors

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

Philipp Helle, Airbus Group Innovations, Germany

VALID 2014

Forward

The Sixth International Conference on Advances in System Testing and Validation Lifecycle (VALID 2014), held between October 12 - 16, 2014 in Nice, France, 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 had the following tracks:

- Testing techniques and mechanisms
- Software verification and validation
- System and feature testing

We take here the opportunity to warmly thank all the members of the VALID 2014 technical program committee, as well as the numerous reviewers. The creation of such a 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 effort to contribute to VALID 2014. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the VALID 2014 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope VALID 2014 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the area of system testing and validation. We also hope that Nice, France provided a pleasant environment during the conference and everyone saved some time to enjoy the charm of the city.

VALID 2014 Chairs

VALID Advisory Chairs

Andrea Baruzzo, Università degli Studi di Udine, Italy
Cristina Seceleanu, Mälardalen University, Sweden
Mehdi Tahoori, Karlsruhe Institute of Technology (KIT), Germany
Mehmet Aksit, University of Twente - Enschede, The Netherlands
Amir Alimohammad, San Diego State University, USA
Hema Srikanth, IBM, USA

VALID 2014 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 Bucharest, Romania
Vladimir Rubanov, Institute for System Programming / Russian Academy of Sciences (ISPRAS), Russia
Tanja Vos, Universidad Politécnica de Valencia, Spain

VALID 2014 Industry Chairs

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

VALID 2014

Committee

Table of Contents

Adaptive Knowledge-Supported Testing: An Approach for Improving Testing Efficiency

Philipp Helle and Wladimir Schamai
Airbus Group Innovations
Hamburg, Germany
Email: {philipp.helle,wladimir.schamai}@eads.net

***Abstract*—This paper introduces a new method for automatic test parameter generation that has been named adaptive knowledge-supported testing. The approach uses a combination of random testing for test parameter generation and machine learning and data mining techniques to optimize these test parameters based on the results from previous tests. The goal is to enable efficient testing of complex systems which cannot be tested exhaustively anymore due to the huge number of possible input combinations. The paper provides a description of the method and also results from the evaluation of a first proof-of-concept demonstrator that has been implemented to validate the method.**

Keywords*—*Testing, Adaptive Testing, Machine Learning.

I. Introduction

Increasing system complexity results in an increase in complexity of the test engineers' task to ensure that systems are correct. In recent industry practice, the verification phase is commonly the longest phase in system development and is the most critical to completing a product on time [1].

This raises the need for the development of new techniques and methodologies that can provide the test engineers with the means to achieve their goals quickly and with limited resources. These solutions succeed in removing much of the manual labour traditionally involved in the verification process. Tasks such as test execution and test report generation are now typically automated to a high degree. Model-based testing (MBT) is a new trend in industry that focuses on automatic test case generation [2] [3]. However, test parameter generation is often still a manual task [4].

Adaptive knowledge-supported testing is a new method for automatic test parameter generation. It combines methods from the area of machine learning and artificial intelligence, e.g., neural networks and methods from the area of data mining, e.g., data clustering and existing methods for test parameter generation, e.g., random testing.

Under the assumption that critical test points, i.e., stimulus combinations that lead to errors, typically occur in groups in the whole test parameter space, adaptive knowledge-supported testing allows to generate optimized test parameters from previously executed tests and their results.

This paper is structured as follows: Section 2 provides some background regarding testing and machine learning. Section 3 deals with related and prior work. Section 4 introduces the adaptive knowledge-supported testing method using a running example and, finally, Section 5 concludes the paper.

II. Background

A. Testing

"Testing is the process of executing a program with the intent of finding errors" [5]. Testing is a well-known discipline in the software and system engineering fields, and in recent decades many testing strategies have been developed, such as stress testing, fault injection, coverage-based testing, black-box, and white-box testing, or combinations of these.

However, it has been pointed out that "[in] general, it is impractical, often impossible, to find all the errors in a program" [5] and "every testing method (save exhaustive testing [..]) is less than perfect" [6]. Testing cannot guarantee the absence of errors, but it can help to discover their presence. The economics of testing, i.e., the balance between the testing effort and project time or resource constraints, depends on the selected testing strategy and the way test cases are designed, as well as the experience of the testers [5].

Half of all embedded systems development projects are way behind schedule and less than half of the designs meet 20% of the expectations in terms of functionality and performance according to the study in [7]. This is despite the fact that around half of the total development effort is spent on testing [7], [8]. These numbers underline the importance and desirability of reducing test effort by advances in the testing methodologies, especially considering the trend for "increase in software complexity [and] shorter innovation cycle times" [9].

B. Machine learning

According to a standard definition, "Machine learning is programming computers to optimize a performance criterion using example data or past experience." [10]. And more formally: "a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [11].

Transferred to our application this translates as follows: The adaptive knowledge-supported testing learns from *past test results* (experience E) with respect to the task *test parameter generation* (task t) and performance measure *effectiveness of the generated test parameter sets in detecting errors* (performance measure p). So, the *test parameter generation*, measured by the *effectiveness of the generated test parameter sets* improves with the *number of past test results*.

There are different machine learning techniques, such as supervised, unsupervised or reinforcement learning, as well as classification methods, such as decision trees, naive Bayes classifier, support vector machines, neural networks, etc. The methods differ in learning complexity, classification accuracy and robustness, the possibility to interpret the generated results, and performance. Each of the classification methods has advantages and disadvantages. For example, decision trees may be computationally expensive because of the number of distinct nodes to be created. Support vector machines require more effort in setting up the learning and may also be computational expensive. The advantage of using naive Bayes models is the simplicity of learning. However, the underlying assumption is that all input values are independent, which is not always applicable to our setting, in which test parameters values may be interrelated. In turn, artificial neural networks are an alternative to dynamic and non-linear problems because they are not restricted in terms of normality, independence of input data etc.

Artificial Neural Networks (ANN) mimic the biological neural networks in their learning function. ANN are composed of connected neuron models. Each connection has a different weight that is adjusted throughout learning. A special type of artificial neural networks is the Probabilistic Neural Network (PNN). It is a four layers feed-forward network proposed by Specht [12]. When using the Dynamic Decay Adjustment (DDA) algorithm [13] it is possible to build the network dynamically based on the numerical training data. The output of the trained network are inferred rules that enable to predict the probability that new test data belongs to a certain target category. We used PNN (DDA) for our case study for learning from previous test runs and producing new test sets by predicting (i.e., selecting new test data based on the their highest probability).

III. Related Work

Random testing, i.e., random selection of test cases, is generally regarded as not only a simple but also an ”intuitively appealing” [14] technique amongst the black box techniques for test case generation. In random testing, test cases may be randomly chosen based on a uniform distribution or according to other distributions that are inferred from the operational profile of a unit under test (UuT). Hamlet [6] points out that the main benefits of random testing include the availability of efficient algorithms to generate test cases, and also its ability to provide reliability and statistical estimates. Using random test inputs allows many design requirements to be verified very quickly with minimal manual effort. Random tests also have the additional possible benefit of generating test cases that human test engineers would not necessarily think of [15]. Studies have shown that systematic testing methods are not much better at finding failures than random testing [16] and more recent research ”further support[s] the use of random testing in real-world software”[17].

However, random testing also has weaknesses, e.g., ”a vast number of test points are required” [6] and knowledge of the operational profile of a UuT are required to infer suitable distributions for the random number generators. Also, random testing usually does not produce all test cases that are needed to verify a design. The test engineer must evaluate the coverage results of the executed tests and determine, which cases remain to be tested, which can then either be written manually or generated by adjusting the random number generator in an attempt to steer the random test generation into the untested scenarios.

Several new approaches try to combine random testing with a more systematic approach to get the best of both worlds: automatic and quick test case generation coupled with a system to steer the test case generation.

Adaptive Random Testing (ART) is a method based on random testing that seeks to distribute test cases more evenly within the input space [18]. It uses two separate sets of test cases, the executed set and the candidate set, which is a set of randomly generated test points. At each iteration one or more test points are selected from the candidate set and used for a test. The criterion for selection is maximum distance from previously executed tests, which results in a more even spread of test cases in the test space. The distance function needs to be defined for each type of test. The example in the paper uses the Euclidean distance. Adaptive knowledge-supported testing also distinguishes between the executed set of tests and the candidate set of tests, which is generated using random methods. Instead of maximising the distance between test cases to evenly spread the tests we use machine learning to focus testing.

Coverage Directed Test Generation (CDG) uses coverage measurement together with a random test generator in order to assess the progress of the testing process [15]. The coverage analysis allows to modify the directives for the test generators and thereby to target areas of the UuT that are not covered well.

More recently, effort has been made to couple CDG with machine learning techniques to close the manual feedback loop from coverage analysis to test parameter generation. Machine learning, i.e., a Bayesian network, is used to observe the impact of input stimulus changes on coverage goals and a subsequent automatic steering of the input generation parameters so that the coverage is maximised [15]. It has been shown that this kind of CDG can successfully generate test directives from an analysis of observed test coverage gaps to guide the testing to completion more quickly [19][20].

CDG requires a detailed insight into the UuT to allow measuring the coverage achieved by testing, which is not easily possible in blackbox testing. In contrast to CDG, which tries to optimize coverage of the UuT, adaptive knowledge-supported testing aims at optimizing the chance to discover errors in the UuT while minimizing the required number of test runs.

IV. Adaptive knowledge-supported testing

A. Running example

The adaptive knowledge-supported testing method was developed using representative example data provided by a simulation model. The test considered was a power interrupt test, which tests the robustness of the unit under test by applying a number of power interrupts. The test parameters that typically characterize the power interrupt test and their value ranges (positive integers) are provided by Table I:

These test parameters are defined by a test engineer based on his knowledge of the UuT and the goal of the test.

Test parameter	*Min value*	*Max value*	*Smallest step size*
Number of interrupts	1	10	1
Interrupt duration (/10ms)	1	20	1
On time duration (s)	1	20	1

TABLE I: TEST PARAMETERS FOR POWER INTERRUPT TEST

Combinatorics dictates that there exist 4000 possible test points using full factorial parameter combination. We use a monitor-based testing approach described in [21] for evaluating a test run. Other automated test verdict generation approaches could be used as well. The test verdict returned by running one set of test parameters can take three distinct values:

- 0: test passed successfully
- 1: warning point, e.g., some values are anomalous but are still within the allowed value range
- 2: error point, a requirement has been violated

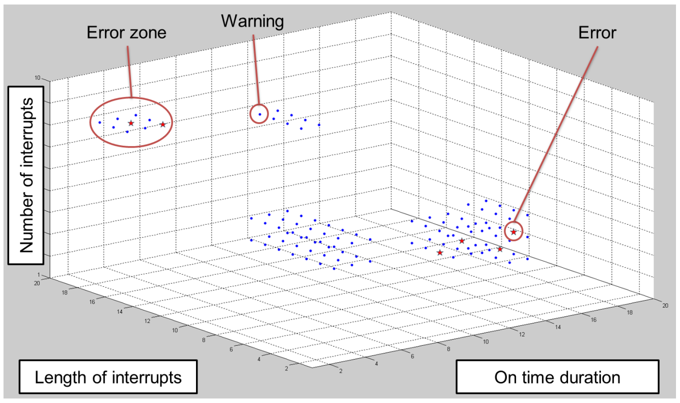


Fig. 1: Running example error distribution

For testing the adaptive knowledge-supported testing method, four error zones have been included in the total test space of 4000 test points, with 6 error points and 96 warning points as Figure 1 shows.

B. Assumptions

One general assumptions is underlying the adaptive knowledge-supported testing approach:

- Warnings and errors occur in groups in the test space: This stems from the observation that errors and warnings do not occur in an isolated fashion in the test space but, since moving from one test point to a neighbouring one represents only a very slight change of inputs, they are found in groups.

Another assumption has been made that is motivated by the running example and is depending on the test evaluation criteria and the possible values of the test verdict.

- Errors are surrounded by warnings: This represents the knowledge from past testing campaigns that testers do not stumble upon errors out of the blue but that, when the stimuli are closing in on an error then a system starts to behave anomalously but still within the bounds of the allowed, e.g., a variable value that has an upper limit starts moving towards this threshold or a variable that should be steady starts to flutter slightly but does not violate the fixed boundaries yet.

Both of these assumptions are based on lessons learnt from past test campaigns and have been confirmed internally by test experts.

C. Process

The starting point for adaptive knowledge-supported testing is always the definition of a test case that is parametrized and of all possible stimuli for the system under test for the given test case. This is done by the test engineer.

For our running example, the abstract parametrized test case can be informally described as follows:

1) Turn on the UuT.
2) After `$OnTimeDuration` start the first power interrupt by turning off the power supply and turning it back on after `$InterruptDuration`*10 ms.
3) If `$NumberOfInterrupts` is greater 1, after `$InterruptDuration`*10 ms initiate the next power interrupt until `$NumberOfInterrupts` interrupts have been executed.
4) Five seconds after the last power interrupt check if the UuT is in the normal operating mode and has successfully passed the initialisation.

The unique benefits of adaptive knowledge-supported testing come to fruition when existing test results are available. If this is not the case then adaptive testing largely corresponds to the underlying test method that is used for the test data generation, i.e., without existing rest results adaptive testing using random data generators becomes random testing.

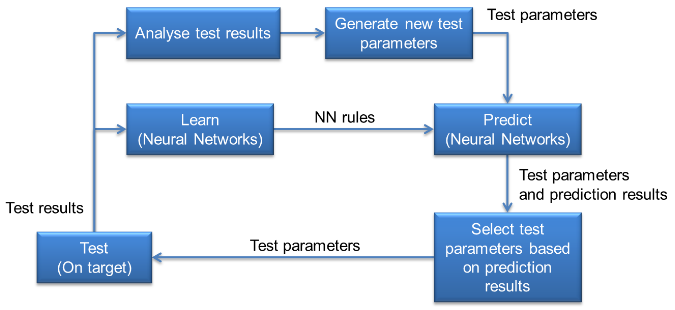

Analyse test results
Generate new test parameters
Test parameters
Learn (Neural Networks)
NN rules
Predict (Neural Networks)
Test parameters and prediction results
Test results
Test (On target)
Test parameters
Select test parameters based on prediction results

Fig. 2: Adaptive knowledge-supported testing process

Figure 2 provides an overview of the general process for the adaptive knowledge-supported testing.

1) **Analyse test results** The goal is to find starting points for the generation of new test parameter values. This may be achieved using different methods:
   - Clustering: test results are grouped into clusters. A possible starting point can then be the center of a cluster.
   - Error point: new starting points are all the discovered error points

2) **Generate new test parameters** Based on the results of the test results analysis and the derived starting points, new potential test parameters are generated to form the candidate set. Various methods can be used for that:
    - Random: new values are generated randomly
    - Stochastics: new values are generated based on stochastic distributions (e.g., Gaussian distribution)
    - Fixed step sizes: new values are generated using steps with predefined step sizes from the starting point(s)

    Other methods for test parameter generation, such as the Category-Partition Method [22] or other structured parameter generation methods could be used as well here.
3) **Learn and predict** Test results from previously executed tests are put into a neural network to derive a decision function. This function can then be used to automatically classify new input data into the three classes that are relevant for our need: test passed successfully, warning and error. New input data means test data that has not been used for learning before. Simply put, the trained neural network is used to predict which of the newly generated test input combinations from the candidate set are highly likely to produce an error or a warning. This information is used in the next step to select a desired set of new test stimuli.
4) **Select test parameters based on prediction results** Usually, test parameter generation results in a very large number of new test parameter combinations, especially when the new individual test parameter values are combined using the Cartesian product to generate new test points. Since it is not always possible to run thousands of tests, this process steps allows reducing the final number of new test points. In this step, some of the test points from the candidate set are selected for the next test execution. This can be done using different selection criteria:
    - Only test points with high error probability (according to the prediction)
    - Mix of test points with high and low error probability
    - Absolute limit for number of test points
5) **Execute test** One by one all the newly selected test points are used to drive one test and obtain a test result. Ideally, this task is automated but depending on the type of task it may also be conducted completely manually. Test execution is not in the focus of this work but has a strong impact on the number of tests that can be conveniently executed. A manual test that lasts one or two hours cannot be conducted a 1000 times with different parameters while a fully automatic test that executes in a couple of seconds can.

Note, that the process is iterative. In each iteration loop, the focus of the test effort is adapted and shifted according to the knowledge gained from the accumulated past results, hence the name of the approach.

Past test results are not necessarily limited to tests on the same version of or indeed the very same type of UuT. The power interrupt test is a very general test, that is applicable to all kinds of components. Different components with a similar start-up routine might exhibit similar failures especially if there are other influencing factors, e.g., in our case two different components might be from the same supplier and therefore use the same kind of power converters, which have a heavy influence on the behaviour reacting to power interrupts or two components might have the same kind of interface, e.g., a CAN bus interface, that is implemented using the same commercial-of-the-shelf interface controller.

Deciding if past test results from another test campaign are suitable for the current UuT is a task left to the test engineer but might, in the future, be supported by a classification of different components, e.g., using an ontology database.

D. Implementation

The adaptive knowledge-supported testing approach has been implemented in a proof-of-concept demonstrator. This demonstrator is based on the Konstanz Information Miner (KNIME) [23] tool, which is available under the GPL GNU Public License, Version 3, an open source license. KNIME is a data analytics platform for data access, transformation, mining and visualisation. It provides a basic set of data processing operations, called nodes, that can be combined graphically in a so-called workflow to achieve complex information manipulation processes.

The test bench and the UuT were implemented as a single simulation, basically a lookup table that accepts a test parameter set at a time and provides the ”test result” as a three-valued integer output as explained before. This allowed us to define the errors zones so that we could benchmark the performance of the adaptive knowledge-supported testing approach. Comma Separated Values (CSV) was chosen as the data exchange format between KNIME and the simulated test bench because it is natively supported by KNIME and an easily adaptable format that can be read in a standard editor, which eases debugging. The complete demonstrator setup is shown by Figure 3.

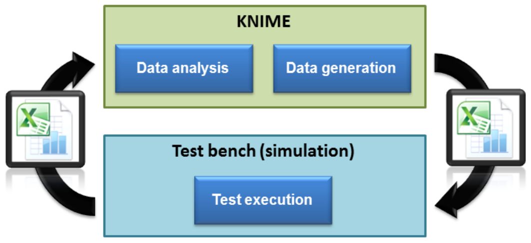


Fig. 3: Adaptive knowledge-supported testing demonstrator

E. Evaluation results

Table II shows the result from the application of the adaptive knowledge-supported testing approach to the power interrupt test. Four iterations of 100 test sets each where conducted. For comparison, we included a brute-force approach, which simple runs all 4000 possible test points and unsurprisingly discovers all warnings and errors but also has a high cost attached to it, i.e., the number of tests per uncovered error or warning is significantly higher than using the adaptive testing approach.

	Method		
Metric	*Brute-force*	*Adaptive testing I*	*Adaptive testing II*
Tested points	4000	400	400
Warnings found	96	22	67
Errors found	6	0	2
Tests/Warnings	41,7	18,2	6
Tests/Errors	666,7	n/a	200

TABLE II: COMPARISON BETWEEN BRUTE-FORCE AND ADAPTIVE TESTING

As already discussed in Section IV-C, the process step ”Select test parameters based on prediction results” permits different options for the selection of the new test parameter sets from the pool of generated test parameters. To understand these options, it is important to understand the output of the predictor. For each test set the predictor has four different outputs:

- **0**: The predicted likelihood between 0 and 1 that this test set will return a 0 result (test passed successfully)
- **1**: The predicted likelihood between 0 and 1 that this test set will return a 1 result (warning)
- **2**: The predicted likelihood between 0 and 1 that this test set will return a 2 result (error)
- **Winner**: Either 0, 1 or 2; the most likely, i.e., the one with the highest likelihood, of the three possible results.

We use the Winner value, as well as the 0 likelihood value for selecting test parameters for the next iteration of testing from the candidate set of test parameter values that were created in the ”Generate new test parameters” process step.

As can be seen from Table II two different options of the adaptive knowledge-supported testing approach have been evaluated:

- **Adaptive testing I**: All the test parameters, for which the predictor predicts a warning or error result (Winner is 1 or 2) are included in the new test set. Additionally, to fill the set up to 100 new test sets per generation iteration, the test points, for which the predictor predicts a 0 test result with the lowest likelihood are included as well.
- **Adaptive testing II**: All the test parameters, for which the predictor predicts a warning or error result (Winner is 1 or 2) are included in the new test set. Additionally, to fill the set up to 100 new test sets per generation iteration, a mix of 50 percent test points, for which the predictor predicts a 0 test result with the lowest likelihood and 50 percent random test points from the remaining points are included.

As we can see from the results, the second run, which includes randomly chosen test points fares better at detecting error and warnings than the run, which focuses on the most likely negative test results. The reason for that is that by focusing only on likely error producing test points the chance to uncover new error zones in the test space is increased. Ultimately, this means that a combination of random testing and focused testing delivers the most promising results.

Table III illustrates the iterative approach of adaptive testing. The results show that adaptive testing is able to exhaustively check an error zone once it is discovered. Using the random element in the ”Select test parameters based on prediction results” process step allows to uncover further error zones.

	Iteration				
Metric	*1*	*2*	*3*	*4*	*5*
Tested points	100	200	300	400	500
Warnings found	2	24	42	67	70
Errors found	0	2	2	2	2
Tests/Warnings	50	8,3	7,1	6	7,1
Tests/Errors	n/a	100	150	200	250

TABLE III: INCREMENTAL USAGE OF ADAPTIVE TESTING

A further evaluation was done comparing the results from the adaptive testing to pure random testing. To establish a mean value for the effectiveness of random testing, Monte Carlo simulations with 10000 runs each were conducted for different numbers of randomly (uniform distribution) selected tested points. Table IV contains the results.

Metric	*Number of tested points*				
Tested points	100	200	300	400	1000
Warnings found (mean)	2,4	4,83	7,19	9,59	24
Errors found (mean)	0,15	0,29	0,45	0,60	1,5
Tests/Warnings	41,7	41,4	41,72	41,71	41,7
Tests/Errors	666,7	689,7	666,7	666,67	666,7

TABLE IV: RANDOM TESTING RESULTS

The first evaluation results look promising. The first iteration of adaptive testing corresponds to random testing as no test results where available to optimise the test parameter generation. After the first iteration, adaptive testing proved to be more effective in detecting warnings and errors than pure random testing. It should be noted, however, that a more thorough evaluation is called for, where especially different error distributions will be evaluated and more runs of the adaptive testing approach need to be conducted to form a more substantiated statement about the overall effectiveness of the adaptive testing approach. One outcome of this work will also be a guideline for the user which supports the selection of the various possible options that our method has. Furthermore, it is planned to evaluate the approach using a real unit under test to show its usefulness in an industrial context.

V. CONCLUSION

In this paper, we present a new approach for testing models or systems. The approach leverages knowledge captured in previous tests in order to minimise the number of required tests for detecting errors. Our first case study shows that this approach is promising because of its ability to locate errors by using a fraction of the number of tests compared to a brute force or random testing approach. This is achieved by a mixture of random samples, which enable discovering new error zones and test points that focus on zones as soon as first error indicators are found.

Our case study shows the first promising results. However, the validity of the approach beyond the presented results is still an open question. It is subject to our future work to experiment with other classification methods, such as support vector machines or naive Bayes classifiers, as well as, evaluating the approach using different sets of randomly generated training data. Moreover, we plan to automate the workflows in KNIME in order to minimize the manual effort for setting up and running the tests.

In addition to that, we plan to run a more thorough evaluation campaign. In this we will use a larger example case with a higher number of possible test input combinations to compare our approach to a number of systematic testing approaches in terms of complexity, runtime and testing efficiency.

REFERENCES

[1] R. S. Pressman, Software Engineering - A Practitioner's Approach, 8th Edition. McGraw-Hill, 2014.

[2] J. Peleska, "Industrial-Strength Model-Based Testing - State of the Art and Current Challenges," ArXiv e-prints, Mar. 2013.

[3] M. Shafique and Y. Labiche, "A systematic review of model based testing tool support," Software Quality Engineering Laboratory, Department of Systems and Computer Engineering, Carleton University, 2010, p. 21.

[4] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Software Testing, Verification and Reliability, vol. 22, no. 5, 2012, pp. 297–312.

[5] G. J. Myers, C. Sandler, and T. Badgett, The art of software testing. John Wiley & Sons, 2011.

[6] R. Hamlet, "Random testing," Encyclopedia of Software Engineering, 1994.

[7] V. Encontre, "Testing embedded systems: Do you have the guts for it," IBM, November, 2003.

[8] A. Helmerich, N. Koch, L. Mandel, P. Braun, P. Dornbusch, A. Gruler, P. Keil, R. Leisibach, J. Romberg, B. Schätz et al., "Study of worldwide trends and r&d programmes in embedded systems in view of maximising the impact of a technology platform in the area," Final Report for the European Comission, Brussels, Belgium, 2005.

[9] P. Liggesmeyer and M. Trapp, "Trends in embedded software engineering," IEEE Software, vol. 26, no. 3, 2009, pp. 19–25.

[10] E. Alpaydin, Introduction to machine learning. MIT press, 2004.

[11] T. M. Mitchell, Machine learning. McGraw-Hill, 1997.

[12] D. F. Specht, "Probabilistic neural networks," Neural Networks, vol. 3, no. 1, 1990, pp. 109–118.

[13] M. R. Berthold and J. Diamond, "Constructive training of probabilistic neural networks," Neurocomputing, vol. 19, no. 1, 1998, pp. 167–183.

[14] L. J. White, "Software testing and verification," Advances in Computers, vol. 26, no. 1, 1987, pp. 335–390.

[15] J. S. Vance, "Application of bayesian networks to coverage directed test generation for the verification of digital hardware designs," Ph.D. dissertation, University of Pittsburgh, 2010.

[16] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," Software Engineering, IEEE Transactions on, no. 4, 1984, pp. 438–444.

[17] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," Software Engineering, IEEE Transactions on, vol. 38, no. 2, 2012, pp. 258–277.

[18] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making. Springer, 2005, pp. 320–329.

[19] S. Fine, A. Freund, I. Jaeger, Y. Mansour, Y. Naveh, and A. Ziv, "Harnessing machine learning to improve the success rate of stimuli generation," vol. 55, no. 11. IEEE Transactions on Computers, 2006, pp. 1344–1355.

[20] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in Proceedings Design Automation Conference. IEEE, 2003, pp. 286–291.

[21] P. Helle and W. Schamai, "Towards an integrated methodology for the development and testing of complex systems," in VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle, 2013, pp. 55–60.

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

[23] M. R. Berthold et al., "KNIME: The Konstanz Information Miner," in Data Analysis, Machine Learning and Applications, ser. Studies in Classification, Data Analysis, and Knowledge Organization, C. Preisach, H. Burkhardt, L. Schmidt-Thieme, and R. Decker, Eds. Springer Berlin Heidelberg, 2008, pp. 319–326.

Test-Driven Agile Simulation for Design of Image Processing Systems

Anna Yupatova, Vitali Schneider, Winfried Dulz, Reinhard German
Friedrich-Alexander-University of Erlangen-Nürnberg, Germany
Department of Computer Science 7
{anna.yupatova, vitali.schneider}@fau.de {dulz, german}@cs.fau.de

***Abstract*—Image processing systems are characterized by very high computational demand caused by large amounts of data, short response times, and the complexity of image processing tasks. For these reasons, specialized hardware solutions based on multiple processing cores, complex interconnects, or custom hardware elements are used for image processing. Due to the complexity of the computational tasks, the complexity of the specialized hardware solutions is continuously increasing. Therefore, new design techniques that reduce the risk of lacks in the system design or of expensive design-to-implementation iterations are desirable. Test-driven Agile Simulation (TAS) is a general-purpose approach that combines novel model-based simulation and testing techniques to achieve an improved overall quality for the development process. In this paper, we present an application and extension of the TAS approach for the efficient design process of image processing systems.**

Keywords–Test-driven agile simulation, model-based engineering, UML, SysML, MARTE, UTP, image processing systems

I. Introduction

Image processing systems (IPS) play an increasingly important role in our daily life with applications for medical diagnosis, remote sensing, or fingerprint recognition. In general, image processing tasks have very high computational demands. Due to continuously changing requirements, decreasing times and physical restrictions, IPS are becoming more heterogeneous resulting in the distribution and deployment of computational tasks on different processors and programmable logic units ([1][2]). Without the usage of effective design tools and development techniques, the realization of a complex heterogeneous IPS is difficult to carry out. This leads to a decreasing quality of the development process and finally to deficient systems.

Test-drive Agile Simulation (TAS) is an agile approach that improves the overall quality of the development process and reduces the design and development costs, while the reliability of possibly complex implementations increases due to early validation techniques. The main focus of TAS is on constructing the models that allow to detect design errors or inconsistencies in a system specification as early as possible by simulating the given system and executing test cases at the model level. In our work, we deploy the *VeriTAS* [3] framework that supports the automation of consecutive steps of the TAS approach. Thus, the simulation and testing are applied in earlier stages of a development process which also increases it's agility.

To construct models, TAS provides a modeling methodology [4] based on the UML (Unified Modeling Language) and applies multiple extension profiles, such as System Modeling Language (SysML) [5], Modeling and Analysis of Real-time and Embedded systems (MARTE) [6], and UML Testing Profile (UTP) [7]. Due to the utilization of a general-purpose modeling language, the provided modeling methodology is not limited to a specific application domain. However, in the context of heterogeneous image processing systems, one has to deal with typical image processing peculiarities and issues like modeling of image processing pipelines or the distribution of computational tasks on different hardware components. In this paper, we will explain the utilization of the TAS approach in the context of heterogeneous image systems.

II. Overview of the general design flow

The TAS approach structures the design process into the modeling of requirements, of high-level and refined system specifications as well as of test specifications using UML and a set of extension profiles (see Figure 1). Starting with the

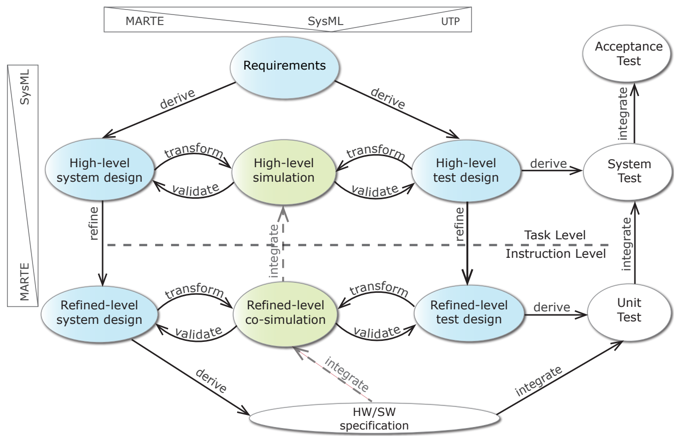


Figure 1: General design flow of the TAS approach.

common requirements specification, our approach is deriving system and test models in the subsequent steps independently from each other in order to ensure their mutual validation. In the high-level specification phase, the modeling is focused on the functional description of a system and corresponding tests. The functional models may be enriched in the subsequent refined specification phase with finer implementation or technical details of the functionality or of additional components for the hardware/software co-design.

Based on the developed models, our approach supports functional verification and performance analysis using simulation-based techniques at different abstraction levels. Prior to the expensive implementation and testing on the real hardware, the simulation of modeled system as well as the simulated execution of test cases allow the validation of the designed system with the specified tests at more abstract levels. Defects found in system specifications during an agile simulation step are easier to correct in the models than in the derived source code and corresponding test scripts.

III. Overview of the Design Flow for Image Processing Systems

The general modeling methodology for TAS is based on a combined subset of SysML, MARTE and UTP. However, the standard UML profiles do not provide enough semantics for the effective design process in the image processing domain. In order to introduce the required semantics in a specific application domain, the definition of a Domain-Specific Language (DSL) offers a feasible solution. However, the usage of DSL involves the additional effort to learn a new modeling language as well as to design and to maintain domain-specific model editors. Therefore, we are working on an UML-based library for image processing systems modeling named Lib4IPS. The library provides a pre-characterization of usable components for building models in the image processing domain. Starting with a detailed analysis of the IPS domain, we extract the most important characteristics and major tasks of image processing applications. The common procedure scheme in such applications is called image processing pipeline and is based on a similar workflow, namely: (1) image acquisition, (2) image pre-processing using local operators, (3) image processing using global operators, and finally (4) image post-processing using complex operators. For example, local pre-processing operators are used to perform an image filtering like the smoothing or edge detection [8]. In addition, the operators can be divided into groups depending on specific criteria like linear or non-linear filtering. According to this structure Lib4IPS performs the classification and characterization of typical image operations by means of UML. For each operator in the image processing pipeline, we define metrics that are important for application design and for performance analysis issues in the later steps of the development process. These metrics include computation cycles, number of instructions, and memory usage. Furthermore, the library holds specific image processing elements, i.e.: bitmap, vector graphics, pixels or image resolution.

Early validation and testing of system properties may have a deep impact on the performance of the final implementation. Hence, we utilize the general design flow of the TAS approach (see Section II) and adjust it for the IPS design and development process. Based on the requirements, a designer needs to devise system design at a high abstraction level, as illustrated in Figure 2. This level is independent of any hardware platform and application details. Its sole purpose is to describe the systems's functional behavior and to provide an initial performance validation step. The refined-level system design distinguishes between the application, the hardware architecture, and a possible mapping step. The hardware architecture model defines platform resources and captures resource acquisitions, performance and timing constraints. The application model describes the functional behavior of the image processing application in an architecture-independent manner. The allocation relationships involve the mapping of the application model onto the hardware architecture model, after which the system model is able to be validated quantitatively. For validation and testing purposes, we combine two simulation frameworks - *OMNeT++* [9] and *SystemC* [10]. The first simulation framework facilitates the system simulation at a high-level of abstraction with the focus on component's communication interfaces. The second simulation framework enables the simulation of image processing application on hardware components. Iterative simulation and test on the model level supports the planning and customization of system specification in order to achieve a desired quality. Furthermore, the approach helps identifying architectural bottlenecks, taking well-founded design decisions as well as finding ways for optimization of the application's execution efficiency.

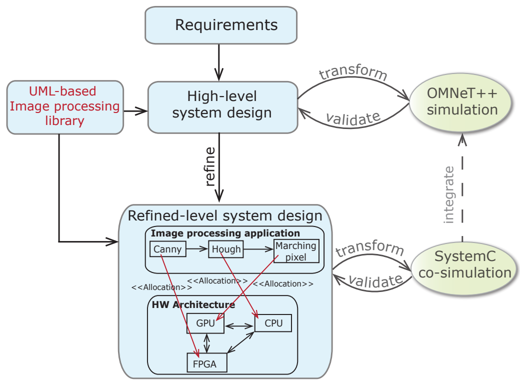


Figure 2: Design flow for image processing systems.

IV. Conclusion and Future Work

In this paper, we explain the application of the TAS approach for the image processing domain that assists a designer in the development of models for the system specification and analysis purposes. We extend the TAS approach by implementing the specific UML-based library which provides required semantics encountered in the image processing domain. In one of our next research steps, we are going to design a result back-tracing strategy for our approach, that will help developers to get simulation results directly into their designed models.

References

[1] D. G.Bailey, "Design for embedded image processing on FPGAs." John Wiley&Sons (Asia), 2011.

[2] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. W. Kim, "Design and Performance Evaluation of Image Processing Algorithms on GPUs," IEEE Transactions on Parallel and Distributed Systems, 2011, pp. 91–104.

[3] A. Djanatliev, W. Dulz, R. German, and V. Schneider, "VeriTAS - A Versatile Modeling Environment for Test-driven Agile Simulation," in Proc. of the 2014 Winter Simulation Conference, Phoenix, AZ, USA, December 2011, pp. 3657–3666.

[4] V. Schneider, A. Yupatova, W. Dulz, and R. German, "How to Avoid Model Interferences for Test-driven Agile Simulation based on Standardized UML Profiles," in Proc. of the Symposium on Theory of Modeling and Simulation, Tampa, FL, USA, April 2014, pp. 540–545.

[5] F. Sanford, M. Alan, and S. Rick, "A practical guide to SysML." Morgan Kaufmann, 2012.

[6] S. Bran and S. Gerard, "Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE." Morgan Kaufmann, 2014.

[7] Object Management Group (OMG), "UTP: UML Testing Profile 1.2." [Online]. Available: http://omg.org/spec/UTP (retrieved October 2014)

[8] M. Petrou and C. Petrou, "Image processing: The Fundamentals," John Wiley&Sons Ltd, 2011.

[9] "OMNeT++ Network Simulation Framework." [Online]. Available: http://www.omnetpp.org/ (retrieved October 2014)

[10] T. Grotker, S. Liao, G. Martin, and S. Swan, "System Design with SystemC." Kluwer Acad. Publ., 2004.

A Testability Transformation Approach for Programs with Assertions

Ali M. Alakeel
Department of Computer Science
University of Tabuk
Tabuk, Saudi Arabia
e-mail: alakeel@ut.edu.sa

Abstract— **Assertion-Based software testing has been shown to be effective in detecting program faults as compared to traditional black-box and white-box software testing methods; however in the presence of large numbers of assertions this approach may be very expensive. As reported in the literature, Assertion-Based software testing executes the whole program based on a given input data in order to find an assertion's violation. Executing the whole program for every assertion may be very costly especially for large programs with very larger number of assertions. The cost is related to search time required during the process of generating test input data to violate such large number of assertions. This paper introduces a testability transformation approach based on the analysis of control and data flow dependencies that affect the execution of every assertion in the program. It achieves this by eliminating program statements that do not lead the program flow control to the assertion under consideration. A small case study is presented, which demonstrates the value of the proposed approach.**

Keywords-assertion-based software testing; testability transformation; software testing; data dependency analysis

I. INTRODUCTION

Software testing is the process of executing a program with the intent of detecting faults [1]. Software testing is a very labor intensive and tedious task. For this reason, many studies have been devoted to the automation software testing [2]-[7]. There are two main approaches to software testing: Black-box and White-box [1]. Test data generation is the process of finding program input data that satisfies a given criteria. Test generators that support black-box testing create test cases by using a set of rules and procedures; the most popular methods include equivalence class partitioning, boundary value analysis, cause-effect graphing [1]. White-box testing is supported by coverage analyzers that assess the coverage of test cases with respect to executed statements, branches, paths, etc. Programmers usually start by testing software using black-box methods against a given specification. By their nature, black-box testing methods might not lead to the execution of all parts of the code. Therefore, this method may not uncover all faults in the program. To increase the possibility of uncovering program faults, white-box testing is then used to ensure that an acceptable coverage has been reached, e.g., branch coverage.

Assertion-based software testing [9]-[10] has been shown to be effective in detecting program faults as compared to traditional black-box and white-box software testing methods. The main objective of assertion-based testing is to find a program input on which an assertion is violated. If such an input is found then there is a fault in the program. Some programming languages support assertions by default, e.g., Java [21] and Perl [22]. For languages without built-in support, assertions can be added in the form of annotated statements. In [9], assertions are represented as commented statements that are pre-processed and converted into Pascal code before compilation. Many types of assertions can be easily generated automatically such as boundary checks, division by zero, null pointers, variable overflow/underflow, etc. Therefore, programmers may be encouraged to write more assertions in their programs in order to enhance their confidence in their programs.

As reported by Korel and Al-Yami [9], assertion-based software testing searches for a program input data that may lead to the violation of a given assertion. In order to test whether this input data will violate the given assertion or not, assertion-based testing executes the whole program based on based on the given input data. The process of executing the whole program for every assertion may be very costly in larger programs with possibly very large number of assertions. Therefore, the performance of assertion-based software testing may be degraded. In order to alleviate this problem and to enhance the performance of assertion-based software testing in the presence of larger number of assertions, the main goal of this paper is to utilize the advantages offered by testability transformation (TeTra) techniques [8] during the process of assertion-based software testing.

The approach presented in this paper applies testability transformation techniques [8] on an original program P_o with assertions to produce a new version P_n such that assertion-based software testing will be more effective in testing the new version P_n than it would be in testing the old version P_o. The primary contributions of this paper are: (1) It introduces a new testability transformation mechanism for programs with assertions. (2) It empowers assertion-based software testing approach and makes more effective in large commercial software with very large number of assertions. (3) The approach may be generally applied to programs with complex pre/post conditions or temporarily embedded pieces of code during instrumentation.

The rest of this paper is organized as follows. A background of assertion-based software testing is presented in Section II. In Section III, related work is discussed. The proposed approach is presented in Section IV. A case study to demonstrate the proposed approach is presented in Section V. Conclusions and future work is discussed in Section VI.

II. Assertion-Based Software Testing

Assertions have been recognized as a powerful tool for automatic run-time detection of software errors during testing, debugging, and maintenance [9]-[14]. An assertion specifies a constraint that applies to some state of a computation. When an assertion evaluates to a false during program execution, there exist an incorrect state in the program. An approach which employs program assertions for the purpose of test data generation was presented in [9]. In that research, it was shown that assertion-based testing was able to uncover program faults which were uncovered by black-box and white-box testing. Given an assertion *A*, the goal of Assertion-Based testing is to identify program input for which *A* will be violated. The main aim of Assertion-Based software testing is to increase the developer confidence in the software under test. Therefore, Assertion-Based software is intended to be used as an extra and complimentary step *after* all traditional testing methods have been performed to the software. Assertion-Based Testing gives the tester the chance to think deeply about the software under test and to locate positions in the software that are very important with regard to the functionality of the software. After locating those important locations, assertions are added to guard against possible errors with regard to the functionality performed in these locations

An assertion may be described as a Boolean formula built from the logical expressions and from the (**and**, **or**, **not**) operators. There are two types of logical expressions: Boolean expression and relational expression. A Boolean expression involves Boolean variables and has the following form: e1 *op* e2, where e1 and e2 are Boolean variables or true/false constant, and *op* is one of $\{=, \neq\}$. Relational expressions, on the other hand, have the following form: e1 *op* e2, where e1 and e2 are arithmetic expressions and *op* is one of $\{<, \leq, >, \geq, =, \neq\}$. For example, (x < y) is a relational expression, and (f = false) is a Boolean expression.

The goal of assertion-based test data generation [9] is to identify program input on which an assertion(s) is violated. Assertion-based testing is based on goal-oriented testing [2][15], which requires the execution of the program during the process of test data generation. This method reduces the problem of test data generation to the problem of finding input data to execute a *target* program's statement s. In this method, each assertion is eventually represented by a set of program's statements (nodes). The execution of any of these nodes causes the violation of this assertion. In order to generate input data to execute a target statement s (node), this method uses the chaining approach [15]. Given a target program statement s, the chaining approach starts by executing the program for an arbitrary input. When the target statement s is not executed on this input, a fitness function [4][5][20] is associated with this statement and function minimization search algorithms are used to find automatically input to execute s. If the search process can not find program input to execute s, this method identifies program's statements that have to be executed prior to reaching the target statement s. This way, this approach builds a chain of goals that have to be satisfied before the execution to the target statement s. More details of the chaining approach can be found in [20]. As presented in [9], each assertion is written inside Pascal comment regions using the extended comment indicators: (*@ assertion @*) in order to be replaced by an actual code and inserted into the program during a preprocessing stage of the program under test. Figure 1 shows a sample program with two assertions A_1 and A_2.

```
program example;
var data: array[1..40] of integer;
var x, i, MAX: integer;
var positive: boolean;
begin
1  input(i, MAX, x);
2  positive:= true;
3  data[i]:= x;
4  while i <= MAX do begin
5     Input(x);
6     i:=i+1;
7     data[i]:= x;
8      if (x ≥ 0) then  begin
9         value:= data[i];
10        write('Value entered: ', value);
       end
       else
       begin
11       value := data[i];
12       write('Value entered: ', value);
13        i:= i-1;
14        positive:= false;
       end;
  (*@  (i ≥1) and (i ≤ 40)  @*)                    A1

15     if ((x<0) OR (i=MAX)) AND ((i=MAX)
               OR (positive=false)) then
         begin
16         write(i, MAX, positive);
17         if (i=MAX) OR (positive=false) then
           begin
 (*@ ((i=MAX) or (positive=false)) @*)     A2
18, 19       if (i=MAX) then writeln('Full capacity reached!')
20           else writeln('Negative value entered!');
            end;
          end;
21        positive:= true;
         end;
      end.
```

Figure 1. A Sample program with assertions

Assertion-based software testing [9]-[10] is a promising approach in terms of finding programming bugs. However, this approach may be expensive in terms of search time required to violate each assertion imbedded in the program. This is because this approach is an execution-based approach [2], which depends on finding a program input data that may lead to the violation of an assertion during the program execution on this input data. The problem arises in big size programs with large number of assertions, where the process of re-executing the program for each assertion may be very costly. In order to make assertion-based software testing [9] more effective and efficient in testing big programs with large number of assertions, we propose applying testability transformation [8] on programs with assertions prior to the process of assertion-based software testing.

III. RELATED WORK

Testability transformation (TeTra) is a source-to-source program code transformation with the objective to make the new programs easier to test [8]. In other words, testability transformation seeks to improve the process of test data generation and makes it more successful. Testability transformation approaches have been applied on many types of programs with encouraging results. For example, in [17], testability transformation improved the performance of Evolutionary Testing (ET) [18] for state-based programs. Korel et al. [19] presented a testability transformation mechanism that is based on data dependencies analysis. In this approach a transformation function is constructed for those program statements that need to be considered during test data generation. Then, the process of test data generation is performed on this transformation function instead of the original program. Although the testability approaches presented in [17] and [19] work well for single program statements they cannot be applied directly for programs with assertions because assertion each assertion may be comprised of more than one program statements as will be shown later in the next section. In order for the approach presented by Korel et al. [19] to be applied on programs with assertions, we need to perform a testability transformation for each assertion found in the program.

IV. THE PROPOSED APPROACH

The main objective of this paper is to present a testability transformation mechanism for programs with assertions that may makes assertion-based testing more cost-effective and efficient when applied on programs with large number of assertions. Given an original version of a program, P_o, with assertions, the proposed approach works as follows.

At the first stage, this approach performs a pre-processing scan of P_o during which all assertions are identified. At the next stage the approach performs a testability transformation process for each assertion identified at the first stage. The results of this stage is that each assertions is transformed into a set of nodes (program statements), as will be explained later, in such a way that executing any of these nodes is equivalent to the violation of this specific assertion. Then, the proposed approach designates each node as a *target* node and formulates a conditional branch (p,q) and a real valued fitness function associated with this branch [2] such that the execution of node p leads to the execution of the target node.

At this stage the chaining approach presented by Ferguson and Korel [16] is employed during the process of assertion-based test data generation to change the program's flow of execution to lead to branch (p,q) such that target node is executed. Because re-executing the original program, P_o, during the process of assertion-based test data generation [9] is very costly during the attempt to execute target nodes, in the fourth stage, the proposed approach applies the testability transformation presented in [19] on each of the target nodes as follows.

For each branch (p,q) that leads to the execution of a target node, this testability transformation approach [19] uses data dependency analysis [15][20] in order to identify other program statements that may have influence on leading the program flow towards the target node. There exists a data dependency between two program nodes n_j and n_k with respect to a variable v if the following three conditions are satisfied: (1) v is assigned a valued at n_j, (2) v is used at n_k, and (3) there exists a program's execution path from node n_j to node n_k where variable v is not modified.

For each of the target nodes identified in the previous stage, the testability transformation approach [19] constructs a data dependency sub-graph [19] and then based on this sub-graph, only selected nodes of the original program, P_o, is included in a new code sub-routine called the transformation function: TransFunc() [19]. At this stage, the process of assertion-based test data generation is only performed on TransFunc() in order to find program input data to cause the execution of the associated target node under consideration. By doing so, a huge amount of time is saved during the assertion-based test data generation, because re-executing the TransFunc() is much cheaper than re-executing the whole original program P_o in order to find input program data to execute each target node. Furthermore, it has been shown in [20] that using this method of testability transformation empowers the process of test data generation and makes it more efficient.

In order to clarify how the proposed approach works, consider the following classification. Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be a set of assertions found in an original version of a program P_o. For each assertion $A \in \mathcal{A}$, a set of nodes $N(A) = \{n_1, n_2, \ldots, n_q\}$ where $q \geq 1$, is identified during a preprocessing stage of the program under test, where the execution of any node $n_k \in N(A)$, $1 \leq k \leq q$, corresponds to the violation of assertion A. In other words, an assertion A is violated if and only if there exists a program input data **x** for which at least one node $n_k \in N(A)$ is executed. For example, consider the following sample assertion:

(*@ ((x≥y) **or** (x=z)) **and** ((z≠99) **or** (Full=False)) **and** (z≠0) @*)

The set of nodes for this assertion is: $N(A) = \{ n_1, n_2, n_3 \}$ and the code generated is shown in Figure 2.

```
       IF (x < y) THEN
         IF (x ≠ z) THEN
n1          Report_Violation;
       IF (z = 99) THEN
          IF (Full = True) THEN
n2          Report_Violation;
       IF (z = 0) THEN
n3          Report Violation;
```

Figure 2. Code generated for a sample assertion A

In order for an assertion *A* to be violated the search process attempts to generate a program input data **x** that may leads to the execution of *at least* one of n_1, n_2, or n_3.

V. CASE STUDY

To demonstrate how our proposed approach works, consider assertion A_1 in the sample program of Figure 1. In the preprocessing step, assertion A_1 is transformed into the following code:

```
p1 IF  i <1 THEN
n11  write('Assertion A1 Violation!');
p2  IF  i > 40 THEN
n22    write('Assertion A1 Violation!');
```

where nodes n_{11} and n_{22} are the constituents nodes for assertion A_1 such that the execution of either of these nodes causes the violation of this assertion. Now, the objective of assertion-based testing is to generate program input data that causes the execution of at least one of these nodes [9].

```
function TransFunc(in p_size, int st_ids[], int repts[]): real;
var k, j, i of integer;
var x, i, MAX: integer;
begin
    k:=1;
   while k <= p_size do begin
   case (st_ids[k]) of
         1: begin                          { node 1 }
               input(i, MAX, x);
              break;
            end;
        6: begin                          { node 6 }
              i:= i+1;
             for j:=1 to repts[i]-1 do i:=i+1;
              break;
           end;
         13: begin                         { node 13 }
               i:=i-1;
               for j:=1 to repts[i]-1 do i:=i-11;
               break;
             end;
     end;    { case }
   i:=i+1;
   end;   {while}
   TransFunc:=   (1-i); {return fitness function of problem
node}
end; { function }
```

Figure 3. Testability transformation code generated for assertion A1 to replace original program in Figure 1

In order to lead the program's execution flow towards nodes n_{11} and n_{22}, nodes p_1 and p_2 are designated by the proposed approach as *problem* nodes [19]. In order make the process of assertion-based test data generation more efficient, and to avoid re-executing the whole program, the proposed approach applies data dependency based testability transformation approach [19] on the problem nodes p_1 and p_2. For example, the testability transformation code generated for the purpose of generating test data to violate assertion A_1 through the execution of node n_{11} is shown in Figure 3. Note that the code in Figure 3, only includes program statements that has data dependencies [19] with the problem node p_1 with respect to variable *i* which is used at p_1. Also, note that the fitness function constructed for the problem node p_1 is placed at the return statement of TransFunc() [19] in Figure 3.

By applying this method of testability transformation, only small part of the program code is executed during the process of assertion-based testing which makes assertion-based testing more efficient and suitable for programs with large number of assertions. For example, only the code in shown in Figure 3 is executed during the process applying assertion-based testing on node n_{11} of assertion A_1.

VI. CONCLUSTIONS AND FUTURE WORK

In this paper, we presented a novel software testability transformation for programs with assertions. The presented approach builds upon previous methods of testability transformations and utilizes them for the purpose of making assertion-based testing more efficient. The results of applying the proposed approach on programs with large number of assertions may save valuable testing resources during the process of software testing which enhances rapid development of software products. For our future research, we intend to perform an experimental study to evaluate the effectiveness of the proposed approach in various types of commercial software which may contain large number of assertions.

REFERENCES

[1] G. Myers, "The Art of Software Testing," John Wiley & Sons, New York, 1979.

[2] B. Korel, "Automated Test Data Generation," IEEE Transactions on Software Engineering, vol. 16, no. 8, 1990, pp. 870-879.

[3] X. Xiaojun and S. Jinhua, "The Study on an Intelligent General-Purpose Automated Software Testing Suite," Intelligent Computation Technology and Automation (ICICTA) International Conference, May 2010, pp. 993-996.

[4] K. Karnavel, K. and J. Santhoshkumar, "Automated software testing for application maintenance by using bee colony optimization algorithms (BCO)," Information Communication and Embedded Systems (ICICES) International Conference, Feb. 2013, pp. 327-330.

[5] P. Srivastava, and K. Baby, "Automated Software Testing Using Metahurestic Technique Based on an Ant Colony Optimization," Electronic System Design (ISED) International Symposium, Dec. 2010, pp. 235-240.

[6] P. Mitra, S. Chatterjee, and N. Ali, "Graphical analysis of MC/DC using automated software testing," Electronics Computer Technology (ICECT) 3rd International Conference, April 2011, pp. 145-149.

[7] D. Rafi, K. Moses, K. Petersen, and M. Mantyla, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," Automation of Software Test (AST) 7th International Workshop, June 2012, pp. 36-42.

[8] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," Software Engineering, IEEE Transactions on, vol. 30, 2004, pp. 3-16.

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

[10] A. Alakeel and M .Mhashi,"Application of Intelligent Assertion-Based Testing in String Matching Algorithms," American Journal of Scientific Research, No. 65, June 2012, pp. 77-91.

[11] D. Rosenblum, "A Practical Approach to Programming With Assertions," IEEE Trans. on Sofware Eng., vol. 21, no. 1, January 1995.pp. 19-31.

[12] K. Shrestha and M. Rutherfor, "An Empirical Evaluation of Assertions as Oracles," Proceedings of IEEE Inter. Conference on Software Testing, Verification and Validation, 2011, pp. 110-119.

[13] S. Khalid, J. Zimmermann, D. Corney, and C. Fidge, "Automatic Generation of Assertiosn to Detect Potential Security Vulnerabilities in C Program That Use Union and Pointer Types," Proceedings of Fourth Inter. Conference on Network and System Security, 2010, pp. 351-356.

[14] A. Alakeel, “Intelligent Assertions Placement Scheme for String Search Algorithms,” Proceedings of the Second International Conference on Intelligent Systems and Applications, Venice, Italy, April 2013, pp. 122-128.

[15] B. Korel, “Dynamic Method for Software Test Data Generation,” Journal of Software Testing, Verification, and Reliability, vol. 2, 1992, pp. 203-213.

[16] R. Ferguson, R. and B. Korel, “Chaining Approach for Automated Test Data Generation,” ACM Tran. on Software Eng. and Tethodology, vol. 5, no. 1, 1996, pp. 63-68.

[17] A. Kalaji , R. Hierons, and S. Swift, “A Testability Transformation Approach for State-Based Programs,” In IEEE 1st International Symposium on Search Based Software, Windsor, UK , May 2009, pp. 85-88.

[18] P. Mcminn and M. Holcombo, “The State Problem for Evolutionary Testing,” Proc. Genetic and Evolutionary Computation Conference, 2003, pp. 2488-2498.

[19] B. Korel, M. Harman, S. Chung, and P. Apirukvorapinit, “Data dependence based testability transformation in automated test generation,” In 16th International Symposium on Software Reliability Engineering (ISSRE 05), Chicago, USA, Nov. 2005, pp. 245–254.

[20] M. Pezze and M. Young, "Software Testing and Analysis: Process, Principles and Techniques," John Wiley & Sons, 2008.

[21] K. Arnold, J. Gosling, and D. Holmes, “The Java programming language,” vol. 2. Reading: Addison-wesley, 1996.

[22] L. Wall, T. Christiansen and J. Orwant, "Programming Perl,", 3rd Ed., O'Reilly Media, 2000.

Investigation of Opportunities for Test Case Selection Optimisation Based on Similarity Computation and Search-Based Minimisation Algorithms

Eike Steffen Reetz[1,2], Daniel Kuemper[1], Klaus Moessner[2], Ralf Tönjes[1]

[1]Faculty of Engineering and Computer Science, University of Applied Sciences Osnabrück
Osnabrück, Germany
Email: {e.reetz, d.kuemper, r.toenjes} @hs-osnabrueck.de
[2]Centre for Communication Systems Research, University of Surrey
Guildford, United Kingdom
Email: k.moessner@surrey.ac.uk

***Abstract*—Test Case Diversity investigations promise to reduce the number of Test Cases to be executed whereby addressing one of the drawbacks of automated model-based testing. Based on the assumption that more diverse Test Cases have a higher probability to fail, algorithms for distance analysis and search based minimisation techniques can help to enhance the quality of selection. This work discusses the application of Hamming Distance and Levenshtein Distance to compute similarity scores and outlines how Random Search and Hill Climbing can be applied to the problem of group optimisation based on pairwise Test Case similarity scores. The evaluation results, conducted with a test framework for automated test derivation and execution for IoT-based services, indicates that proposed Group Hill Climbing algorithm can outperform Random Search and at the same time utilising less computation time. The inclusion of the sequence-based Levenshtein algorithm shows advantages over the utilisation of the set-based Hamming-inspired scoring methodology.**

***Keywords*–Model-based testing; Test Case Diversity, Data Analysis, Hill Climbing, Optimisation Problem.**

I. Introduction

Model-based testing offers the ability for (semi-) automated Test Case (TC) creation and execution based on machine interpretable software specifications. Concepts to derive TCs out of a test model result in a lot of TCs, which can not be executed within a reasonable time without expensive testing costs [1]. Therefore, it is eminent to identify which TC and which test data should be selected for execution to ensure the best target test coverage within given time and resources. The underlying assumption is that automated test case creation can guarantee a wide test target coverage only by creating partly redundant TCs. As a consequence, the process of selecting test data and TCs, which have the highest possibility to identify failures, is crucial for a successful appliance of model-based testing paradigms for application testing.

The present work follows a similarity investigation approach, which tries to identify the most diverse TCs for test execution based on a pairwise similarity between all TCs. This approach can improve the selection of TCs, if parts of the TCs have redundancies and the removal of these redundant TCs have a lower impact to the fault detection rate than randomly removing TCs. The present work tries to enhance the understanding of the application of distance algorithm to the problem of TC reduction whereby investigating the impact off set and sequence-based distance algorithm. The pairwise calculation of the distances of different TCs results in a NP-hard problem [2] if it comes to the selection of a group of TCs out of all TCs (set cover problem). One of the open questions is, if specific search-techniques can enhance the performance compared to random search with comparable computation time effort. This can only be the case if the distance between TCs are not normally distributed and correlations exist for example for TC neighbours. Within this work concepts of the identification of local optimums as introduced by Hill Climbing are applied to the problem of group optimisation of the TC selection.

The evaluation of the work is based on a finite state machine, which is utilised to create TCs based on the W-Method [3]. The results of the experiments indicate that a proposed Group Hill Climbing algorithm perform better than random search with 100 repetitions and on the same time requires less computation time. Positive impact of Group Hill Climbing is identified in case of the utilisation of Levenshtein, which provides a higher granularity between the computed similarity scores.

The paper is organised as follows. After the related work presented in Section II, the investigated algorithm of distance calculation and group optimisation based on search methodology and their application of the TC problem are described in Section III. Afterwards, the evaluation setup and the example test model is presented and the test process is outlined in Subsection IV-A. In Section IV the evaluation findings are highlighted and the conclusion wraps up the paper in Section V.

II. Related Work

Related concepts to reduce the number of TCs exist based on i) test execution history, ii) test purposes and iii) similarity investigations (other approaches especially for white-box testing exists cf. [4]). TC reduction based on history is one oft the widely used techniques for regression tests [5]. The knowledge which TCs failed in previous versions of the System Under Test (SUT) is utilised to select the TCs that are executed during the current test procedure. Other works extended this approach by categorising SUTs and assuming that there is a correlation between failures that occur at different SUT of the same category [6]. Nevertheless, for this approach it is curial to have a good history base. The assumption of correlation between failures in different versions or SUTs of the same categories has not been proven so far as been valid for all kinds of software (with our best knowledge). One methodology to overcome a missing execution history is based on mutation testing. For example, Zhang et. al. [7] combined mutation testing with test reduction and TC selection/prioritisation. A more straightforward technique is the reduction of the target coverage [8]. By limiting the area of interest it is possible to reduce the number of TCs, but the question remains, which parts are more important than others. The MINTS tool combines coverage, history and costs data for the test case selection [9]. It models the multi-dimensional minimisation problem as a binary Integer Linear Programming (ILP) and can utilise different ILP solvers. The results indicate that this approach can be as good as classical heuristic techniques. Although, the approach is only applicable for regression testing and white-box testing. In advance, it remains unclear how to proceed if no optimal solution exists (e.g contradicting requirements). Other approaches also try to use scenarios or TC weighting mechanisms by including the knowledge of experts [10].

Recent trends in software engineering indicate that search-based optimisation techniques are a promising candidate for several software issues such as requirements, project planning, testing and re-engineering optimisations [11]. Approaches for test case selection (often called prioritisation within this context) include the utilisation of greedy [12] and clustering [13][14] algorithms for white-box testing. The results indicate that cluster-based algorithm can outperform traditional coverage-based TC selection techniques by including human input. Different to these approaches the outlined work follows a model-based testing approach without human interaction within the TC selection process. One of the first researchers who used these search-based approaches for model-based testing was Cartaxo et. al. [15], who tried to select the less similar TCs while maximising the state or transition coverage of the test model with the remaining TCs. Hemmati et. al [16] extended this approach by utilising several similarity functions and minimisation algorithm and applied them to two larger SUTs. The findings indicate that the choice of technique significantly influences the performance. Although, it remains open if the results can be transferred to other SUT and if there is a general setup which is always the best. While the previous results from Cartaxo and Hemmati focus on maximisation of the fault detection rate with the minimum number of TCs, the outlined work focus on validating the average group similarity of the selected TCs. Different to previous approaches the present work focuses on a more precise interpretation of the methodology and tries to make the results better repeatable and comparable to enable a generalisation of results.

III. ALGORITHMS

For the followed TC diversity approach it is necessary to identify the similarity between TCs. Therefore, the starting influence factor is, which information is included into the comparison. Findings from [16] indicate that the best performance can be achieved by including all information present in the test model. To access the distance between two objects in a multidimensional feature space, set-based similarity scoring mechanism can be used. In our case, a TC is not only a group of objects (e.g., states, transitions, input, output), it is also a sequence which could have a different order of these objects compared to other TCs. Therefore, also sequence-based distance algorithm can improve the test case selection. The result of the similarity computation is a similarity matrix which contains the pairwise similarity between all TCs. In the last step, the target number of TCs is selected based on the similarity matrix. It is the goal to find the group of TCs which have the lowest average similarity between the TCs of the selected group. Note, that this is a NP-hard problem [2] and the optimum can therefore not be found in polynomial time. In our example, discussed in Section IV-A, with a total number of n = 132 TCs the maximum number of combination possibilities is reached with a group size of k = 66 ($\binom{n}{k} \approx 3,8 \cdot 10^{38}$ with n=132 and k=66).

The presented work evaluates the performance of one set-based and a sequence-based similarity scoring computation together with a baseline random-search and Group Hill Climbing. The algorithms are outlined within this section, including pseudo code to ensure that the realisation approach can be validated and compared to other approaches.

A. Similarity Score Computation

As a baseline, a Hamming-inspired algorithm is realised according to the work discussed in [16]. The Hamming Distance is an edit-based distance algorithm, which is widely used in the literature. It defines the minimum required operations to transform one string to another with editing operations (delete, insert, substitute) for strings with the same length [17].

```
Input: allowedSymbols
foreach testCase do
    foreach allowedSymbol do
        if testCase contains allowedSymbol then
            occurrenceBit = 1
        else
            occurrenceBit = 0
        end
        Add occurrenceBit to bitOcurrenceStream
    end
end
foreach bitOcurrenceSteam do
    simStream = pairwise XOR of bitOcurrenceStream
    similarity = simStream / length of bitOccurenceStream
end
```

Figure 1. Hamming-based Similarity Scoring Algorithm.

The pseudo code shown in Figure 1, shows the basic steps for this approach. In all discussed similarity scoring mechanism, the algorithm starts with the identification of the allowed symbols. Dependent on the encoding, this could include input, output symbols, states, guards etc. For the sake of simplicity, all experiments are conducted with allowed symbols for input, output and states. Guards have not been taken into account (cf. Section IV-A which describes the example service). Although, if the algorithm extension would use also guards, it would not change the algorithm but would result in a larger symbol space, which consequences in lower similarities between all TCs. For each TC, the occurrence of the individual allowed symbols is identified. Each occurrence is documented with a *occurenceBit* = 1 and 0 if it is not present. Afterwards this *occurenceBit* is added to a *bitOccurenceStream*. For each TC the *bitOccurenceStream* is pair- and bitwise XOR compared to each *bitOccurenceStream* from all other TCs. The resulting XOR distance is then used to count the bits that are 1 and divide them by the length of the *bitOccurenceStream*. The result is stored as the pairwise similarity score. The classical Hamming Distance algorithm is limited to strings with the same length. Since TCs does not necessarily have the same length, the classical algorithm has been altered as proposed by [11]. As a consequence the implemented Hamming inspired algorithm can handle strings with different length but is only set-based and does not take the occurrence order of the symbols into account.

```
Input: SymbolsOfEachCase
foreach pair of testCases do
    foreach symbolNum of FirstTestCase do
        m[symbolNum, 0] = symbolNum
    end
    foreach symbolNum of SecondTestCase do
        m[0, symbolNum] = symbolNum
    end
    foreach symbol of FirstTestCase do
        foreach symbol of SecondTestCase do
            if symbolFirst == symbolSecond then
                m[symbolFirst, symbolSec] = m[symbolFirst -1 , symbol-
                Sec -1]
            else
                m[symbolFirst, symbolSec] = Min (
                m[symbolFirst -1 , symbolSec] + 1,
                m[symbolFirst, symbolSec -1] + 1,
                m[symbolFirst -1 , symbolSec -1] + 1 )
            end
        end
    end
    similarity = 1 - m[numOfSymbolsFirst, numOfSymbolsSec] /maxNum-
    Symbols;
end
```

Figure 2. Levensthein-based Similarity Scoring based on Wagner-Fischer Algorithm [18].

The realisation of the Levenshtein similarity computation is shown in Figure 2. Levenshtein is also an edit-based distance algorithm and is not limited to strings with the same length [17]. Each edit operation (delete, etc.) results in an increasing distance between the compared strings. The Levenshtein algorithm is initiated by the identification of the symbols, which are part of the current pair of TCs. A matrix is created where the first row and column contain an increasing sequence from one to the quantity of symbols with an increment of one. Based on the basic operations of *add, del, modify* the sequence of symbols of the *FirstTestCase* is compared to the sequence of symbols of the *SecondTestCase*. It is identified how many operation steps are required to alter the *FirstTestCase* and reach the second one. The algorithm is based on the Wagner-Fischer algorithm [18] and it always prefers matches over insertions or deletions even if they provide a better score. The last computed matrix element contain the distance between the two TCs. For comparison reason this distance is then converted to a similarity score and normalised to the *maxNumSymbols*. Different to the realisation of Hematie et. al where only matches have been counted, this algorithm is able to identify the distance between two TCs as intended by the Levenshtein algorithm.

B. TC Selection

As the baseline, the random selection of a group of TCs has been implemented. As outlined in Figure 3 it starts with the input of the *Similarity Scores* matrix, all *TCs*, the target number of TCs and the number of trials (*numTrials*). The output of the algorithm is the list of selected TCs for the execution. For each trial, the target number of TCs is selected out of all TCs. Afterwards, the summary similarity between all selected TCs is computed based on the pairwise similarity scores. For each iteration, the computed summary similarity is compared to the lowest previous summary similarity. After the defined number of trials the group of TCs with the lowest summary similarity is selected for the test execution. While this approach finds the best group of TCs if the number of trials is infinity it is the question if other search algorithms exist, which can outperform this simple random methodology by identifying a better group of TCs within the same amount of resources (e.g., computation time).

```
Input: Similarity Scores
Input: Test Cases (TCs)
Input: Trial Numbers (numTrials)
Input: n /* Target number Test Cases                    */
Output: selTCs /* List of Selected TCs                  */
for $trialNum = 0$ to $numTrials$ do
    tmpSelTCs /* list of selected TCs                   */
    tmpSelTCs = randomly select n test cases out of all
    foreach Selected Test Cases do
        rowSim /* Current Row Similarity                */
        rowSim = sum similarity of tmpSelTCs
        sumSim = sumSim + rowSim
    end
    lastSumSimilarity = 0
    if $sumSim >= lastSumSimilarity$ then
        selTCs /* list of selected TCs                  */
        selTCs = tmpSelTCs
        lastSumSimilarity = sumSim
    end
end
```

Figure 3. Random Search Test Case Group Selection.

One promising candidate is Hill Climbing, which is based on the identification of local optimums. Instead of searching for the best possible solution out of all groups of TCs, it can help to find good but not necessarily the best group of TCs (local optimum). Hill Climbing searches for local optimums by looking at neighbour elements. A neighbour is an element which is structural close. From a start point, the algorithm tries to find a better solution by exchanging one element with a neighbour. The algorithm stops if no better neighbour is found (local optimum reached). Hill Climbing approaches differs how the starting point is chosen, how neighbours are defined and how many elements can be reached with one move operation (numbers of direct neighbours) [16].

The following application of a Group Hill Climbing is shown in Figure 4. The first step is equal to the Random Search algorithm with a number of trials of one. Then, for each TC within this group it is investigated if a neighbour TC exists, which lowers the summary similarity between all TCs of the selected group (*sumSim* $<$ *lastSumSimilarity*). The order of the created TCs thereby defines which TCs are neighbours. Therefore, the methodology how the TCs are created is expected to have influences on the performance of the identification of neighbours. The experiments are only conducted with the W-method and further investigations could quantify the influences of different test case creation methodologies. To reduce the computation effort only the variable part of the summary similarity is computed. The variable part of this summary is the pairwise similarity score between the selected group of TCs (without the current TC) to the neighbours of the current TCs (*rowSim*). The neighbour search is stopped as soon as there is a neighbour with worse characteristics than the last one. Therefore, the computation effort is not deterministic and depends on the initial set of TCs and the characteristics of the neighbours (e.g., if the next optimum is nearby).

```
Input: Similarity Scores
Input: Test Cases (TCs)
Input: n /* Target number Test Cases */
selTCs /* list of selected test cases */
selTCs = randomly select n test cases out of all
foreach Selected TC do
    rowSim /* Current Row Similarity */
    rowSim = sum similarity of selTCs
    tmpSelTCs /* temp. list of sel. TCs */
    tmpSelTCs = selTCs - currentTestCase
    rTC /* Right TC of Current TC */
    rTC = currentTestCaseNumber + 1
    lTC /* Left TC of Current TC */
    lTC = currentTestCaseNumber - 1
    rRowSim /* Row Similarity with rTC */
    rRowSim = sum Similarity of rTC to tmpSelTCs
    lRowSim /* Row Similarity with lTC */
    lRowSim = sum similarity of lTC to tmpSelTCs
    if rowSim > rRowSim < leftRowSimiarity then
        laRowSim /* Last Row Similarity */
        laRowSim = rRowSim
        while rowSim > laRowSim do
            laRowSim = rowSim
            rTC = rTC + 1
            rowSim = sum similarity of rTC to selTCs
        end
        tmpSelTCs = selTCs + (rTC - 1)
    else if rowSim > lRowSim < rRowSim then
        search left in the same way as befor for the right sight
    else
        tmpSelTCs = selTCs + currentTestCase
end
```

Figure 4. Hill Climbing Test Case Group Selection.

IV. Evaluation Results

The different methodologies to compute the similarity between the TCs as well as the minimisation algorithm results in several experimentation setups where the different aspects are evaluated individually. The next Subsection will briefly explain the overall test approach and introduces an example finite state machine, which is conducted for the experiment. Afterwards, the different experiments are explained and the results are discussed.

A. Setup

The TC selection concept of the present work is included in a test automation framework for IoT-based services. The framework is capable to create and execute IoT-domain specific TCs. Nevertheless, the presented test derivation concept is not limited to that and will be discussed here on a generalised level. The test creation process can be described as follows: the framework explained in detail in [19] derives a test model out of semantic descriptions of the SUT. Subsequently, the test model is represented by an extended finite state machine. This test model is utilised to derive TCs based on the W-method. Afterwards, the TCs are translated with a template engine into the TTCN-3 language which afterwards can be compiled and executed with the TTworkbench [20].

An example of a state machine with 5 states, 10 transitions, 2 input messages and 2 output messages is implemented and as a result the W-method creates 132 TCs with full state and transition coverage. The example state machine is depicted in Figure 5. It shows a behaviour of a reactive

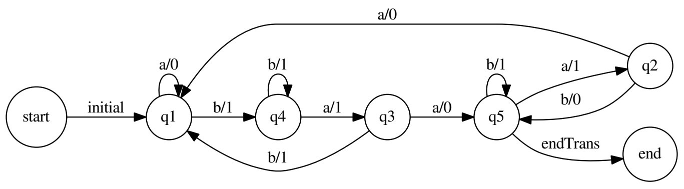


Figure 5. Example Finite State Machine.

system, which will be utilised to evaluate the performance of the similarity scoring and minimisation methodologies within this paper. As an example the input symbols are [*a,b*] and output symbols are [*0,1*]. In order to be executable in our test framework, also transitions for the initialisation (e.g., starting of the SUT) and an end transition exists but does not affect the experiments. The outlined experiments, shown during the next subsection, start after the derivation of the TCs. For each setup, the experiment has been repeated 10,000 times to ensure convincing results. For each combination, of similarity score computation and minimisation algorithm, the experiment is repeated with different target number of TCs from 130 to 5 TCs. Due to computation time the experiments are conducted with a step range of 10 between 130 and 10 target TCs. Although, the results follow a continuous curve and it is not expected to have a divergent behaviour between these values. Each experiment is shown with boxplots to visualise the distribution of the 10,000 repetitions of the experiment. In addition, the median average similarity score is shown in comparison to the other combinations of the algorithms. The time measurements indicate the implemented computation effort, although it does not replace theoretical analysis of the algorithm complexity. The boxplot whiskers show the lowest datum still within 1.5 Inter-Quartile Range (IQR) of the lower quartile and the highest datum still within 1.5 IQR of the upper quartile. Outliers are indicated with a circle.

B. Random Search and Group Hill Climbing with Hamming Similarity Scoring

Figure 6 shows the boxplots of the average similarity between the selected TCs as a function of the number of selected TCs (target TC number). The experiment is conducted with *N=1*, where *N* is the number of trials and with the Hamming Distance inspired scoring methodology. As a general characteristic, the median average similarity decreases while the number of selected TCs also decreases. The diversity of the results increases with the decreasing number of selected TCs due to larger influences of individual TCs, which can result in either small or large average similarity scores. The influence of the number of trials is shown in Figure 7, where the number of trials is N=100. Compared to *N=1* the diversity is much lower and reduces the median average similarity up to 0.067 with the number of selected TCs equals five.

Figure 8 shows the measurements conducted with the Group Hill Climbing algorithm. While the Group Hill Climbing can outperform random search with N=100 during 120 and 30 TCs, it shows limitations for numbers of selected TCs lower than 20. One reason is that the number of possible groups decreases for groups smaller than 66 (for this example) and thus helps the random search approach. As shown in Figure 9 Group Hill Climbing reduces the median average

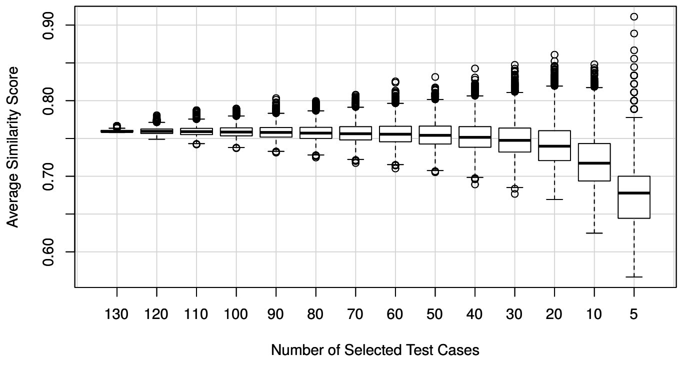


Figure 6. Boxplot of the Average Similarity Between the Selected TCs with Hamming Distance Inspired Scoring and Random Search with N=1.

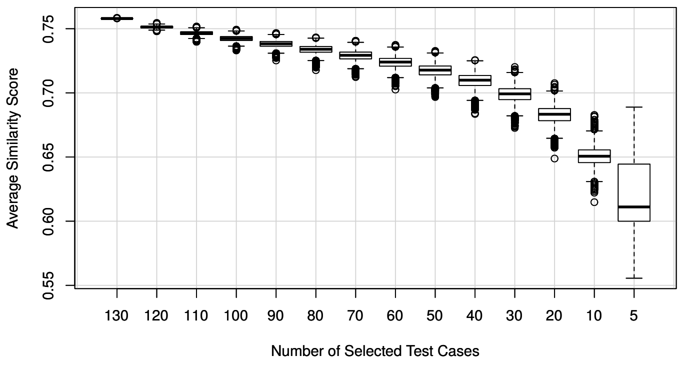


Figure 7. Boxplot of the Average Similarity Between the Selected TCs with Hamming Distance Inspired Scoring and Random Search with N=100.

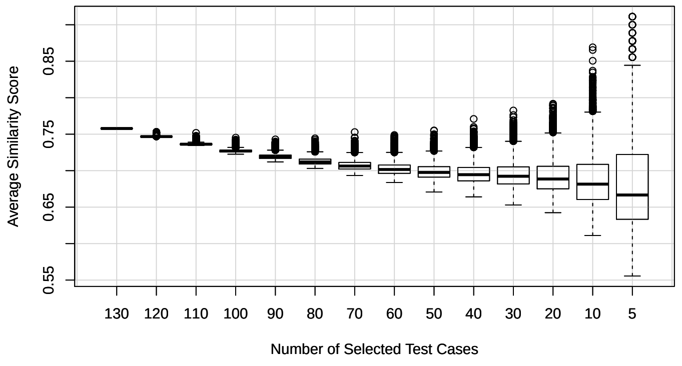


Figure 8. Boxplot of the Average Similarity Between the Selected TCs with Hamming Distance Inspired Scoring and Group Hill Climbing with N=100.

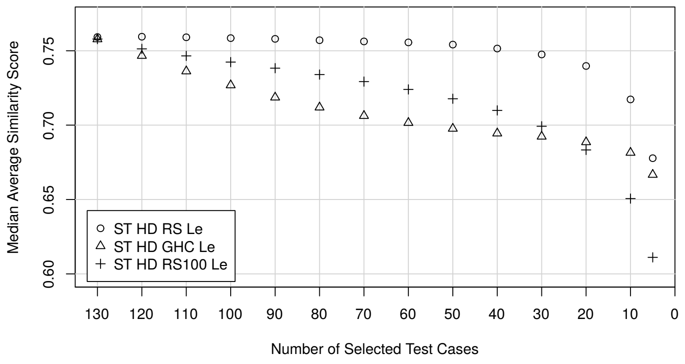


Figure 9. Average Median Similarity Between the Selected TCs with Hamming Distance Inspired Scoring and Group Hill Climbing Compared with Random Search.

similarity compared to Random Search with N=100 up to 0.02 (3,4%) with 70 selected TCs and up to 0,056 (6,6%) compared with Random Search with N=1. One explanation why the outperform is maximised with 60 and 70 TCs selected, is that the number of possible groups is maximised with a group size of 66 ($\binom{n}{k} \approx 3,8 \cdot 10^{38}$ with n=132 and k=66) and therefore random search is very limited. At the same time there are direct neighbours that have not been selected in the initial group and therefore group hill climbing can benefit from the correlation between closely TC neighbours. The computation effort of the implementation is shown in Figure 10. It indicates that Random Search with N=100 requires more computation time than Group Hill Climbing. The Group Hill Climbing computation time decreases with the number of selected TCs since only neighbours for each selected TC are identified. The Group Hill Climbing could be improved further (in terms of computation time), if the Group Hill Climbing tries to find either the group of TC which are not selected (by finding the maximised average similarity) or discover the group of TC which are selected (discover the minimum average similarity). The algorithm can always be applied to the smaller of these two groups and this would optimise the execution time. With this approach it can be assumed that the curve is mirror-symmetric to the right side of 66 selected TCs. Note, that the shown performance is dependent on the derived TCs and further investigations are required to verify this behaviour on different SUTs and different numbers of initial TCs.

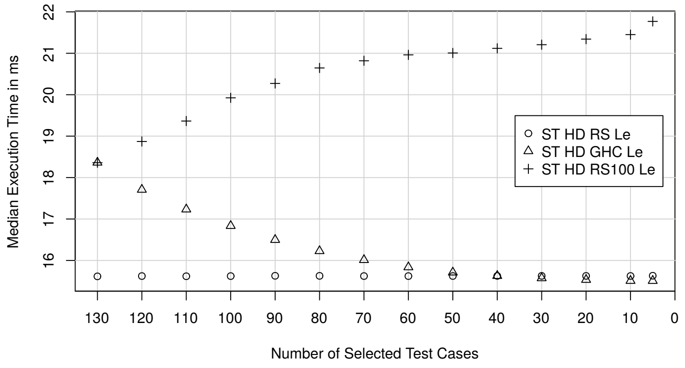


Figure 10. Median Execution Time as a Function of the Number of Selected TCs with Hamming Distance Inspired Scoring and Group Hill Climbing and Random Search.

C. Random Search and Group Hill Climbing with Levenshtein Scoring

As presented before the experiments have been repeated with the Levenshtein similarity scoring discussed in Section III-A. Figure 11 shows the random search with N=1 where N is the number of trials. The results can be compared to the results with Hamming-based Similarity Scoring. While the average similarity with 130 TCs starts with 0,55 instead of 0,76 (due to different normalisations), the trend with a decreasing number of TCs remains the same. The median average similarity decreases, while the diversity increases.

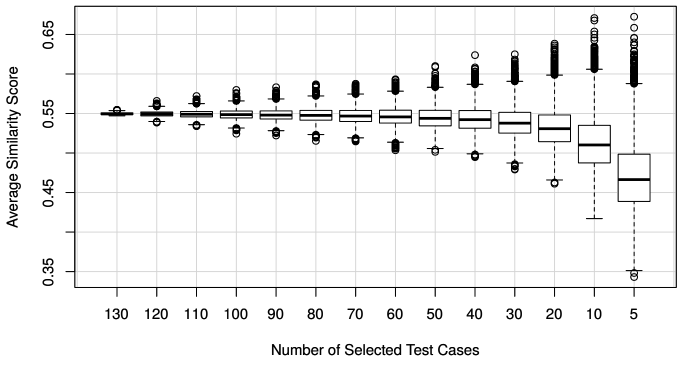


Figure 11. Boxplot of the Average Similarity Between the Selected TCs with Levenshtein Scoring and Random Search with N=1.

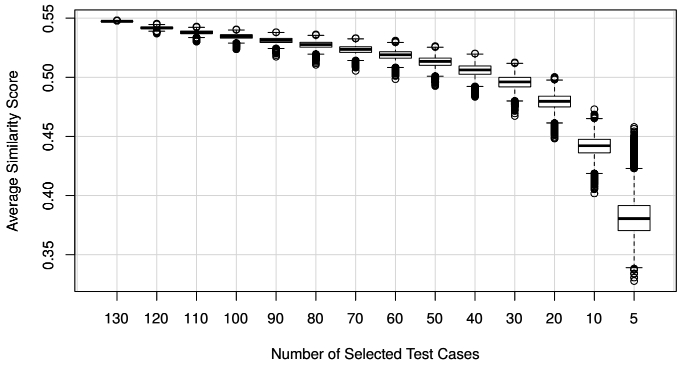


Figure 12. Boxplot of the Average Similarity Between the Selected TCs with Levenshtein Scoring and Random Search with N=100.

Also the enhancement of number of trials, shown in Figure 12 is comparable to the results with Hamming-based scoring. Figure 13 shows the average similarity score as the function of the number of TCs with Levenshtein Similarity Scoring. While the median average similarity decreases with a decreasing number of selected TCs, it exposes also the drawback of Group Hill Climbing, since the diversity of the average similarity score increases. It is much more likely to compute values that are worse than randomly generated with numbers of selected TCs from 20 to 5. As shown in Figure 14 Group Hill Climbing outperforms Random Search (max 0,03 (6%) for N=100 and 0,06 (10,9%) for N=1 for 70 TCs) between 30 and 120 TCs, while using less computation time (Figure 15). Although, the performance gain of Group Hill Climbing is higher with Levenshtein Similarity Scoring, the median execution time is higher compared with Hamming-based Similarity Scoring since the scoring mechanism requires more computation time. Therefore, future work will include the evaluation if Random-search with Hamming-based Scoring, with comparable computation time (more than 100 trials), can outperform the Group Hill Climbing performance with Levenshtein Similarity Scoring.

V. Conclusion

TC reduction based on diversity investigation is a promising approach to enable scalable model-based test automation. To enhance the understanding how distance and search-based minimisation algorithm can be correctly applied to select a

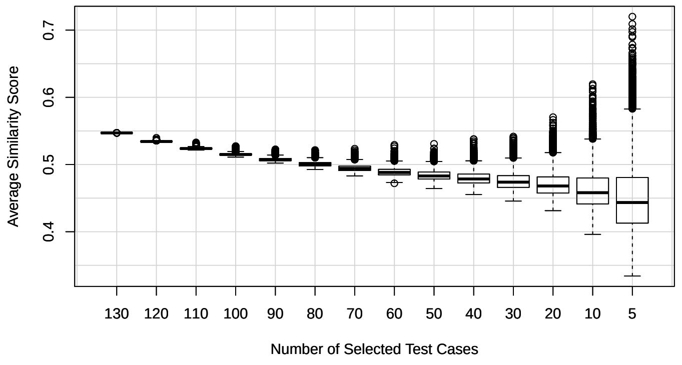


Figure 13. Boxplot of the Average Similarity Between the Selected TCs with Levenshtein Scoring and Group Hill Climbing.

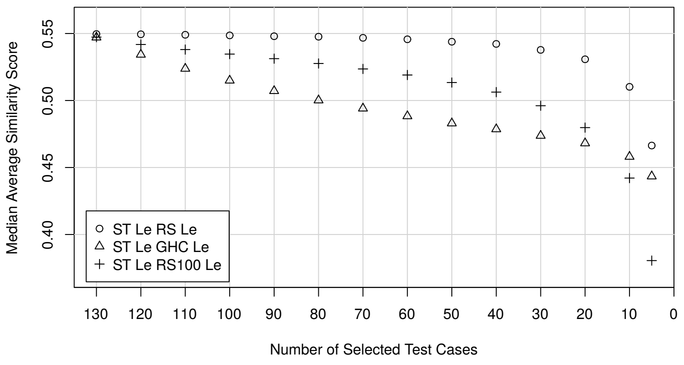


Figure 14. Average Median Similarity Between the Selected TCs with Levenshtein Inspired Scoring and Group Hill Climbing Compared with Random Search.

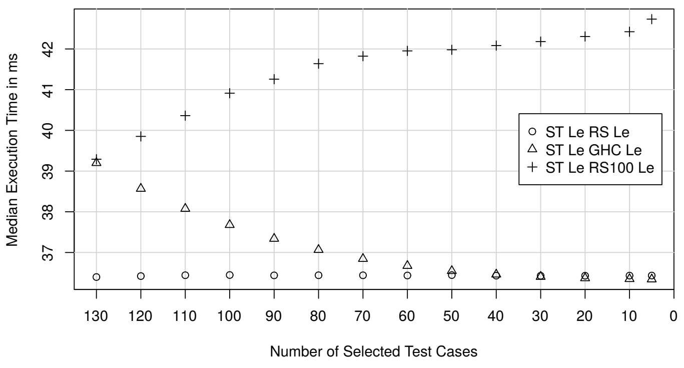


Figure 15. Median Execution Time as a Function of the Number of Selected TCs with Levenshtein and Group Hill Climbing and Random Search.

group of TCs, Hamming-based Similarity Scoring and Levenshtein Similarity Scoring are evaluated together with random search and a proposed Group Hill Climbing algorithm. Based on an example finite state machine, experiments are conducted. The results indicate, that the best performance can be achieved with Group Hill Climbing and Levenshtein Similarity Scoring and on the same time it consumes less computation time compared to Random Search with 100 trials. Future work will include additional search- and similarity computation algorithms, which will be evaluated with different SUTs in order to generalise the results.

VI. ACKNOWLEDGEMENT

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

REFERENCES

[1] E. G. Cartaxo, P. D. Machado, and F. G. O. Neto, “On the use of a similarity function for test case selection in the context of model-based testing,” Software Testing, Verification and Reliability, vol. 21, no. 2, Jun. 2011, pp. 75–100.

[2] A. P. Mathur, Foundations of Software Testing. Addison-Wesley Professional, 2008.

[3] A. Gargantini, “4 conformance testing,” in Model-Based Testing of Reactive Systems (Lecture Notes in Computer Science, vol. 3472), Broy M, Jonsson B, Katoen J-P, Leucker M, Pretschner A (eds). Springer: Berlin, 2005, pp. 87–111.

[4] W. Eric Wong, V. Debroy, and B. Choi, “A family of code coverage-based heuristics for effective fault localization,” Journal of Systems and Software, vol. 83, no. 2, Feb. 2010, pp. 188–208.

[5] E. Engstrm, P. Runeson, and M. Skoglund, “A systematic review on regression test selection techniques,” Information and Software Technology, vol. 52, no. 1, Jan. 2010, pp. 14–30.

[6] W.-T. Tsai, X. Zhou, Y. Chen, and X. Bai, “On testing and evaluating service-oriented software,” Computer, vol. 41, no. 8, Aug. 2008, pp. 40–46.

[7] L. Zhang, D. Marinov, and S. Khurshid, “Faster mutation testing inspired by test prioritization and reduction,” in Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, Jul. 2013, pp. 235–245.

[8] C. Jard and T. Jéron, “Tgv: theory, principles and algorithms,” International Journal on Software Tools for Technology Transfer, vol. 7, no. 4, Aug. 2005, pp. 297–315.

[9] H.-Y. Hsu and A. Orso, “Mints: A general framework and tool for supporting test-suite minimization,” in Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE, May 2009, pp. 419–429.

[10] F. Basanieri, A. Bertolino, and E. Marchetti, “The cow_suite approach to planning and deriving test suites in uml projects,” in UML 2002The Unified Modeling Language. Springer, Sep. 2002, pp. 383–397.

[11] M. Harman, S. A. Mansouri, and Y. Zhang, “Search based software engineering: A comprehensive analysis and review of trends techniques and applications,” Department of Computer Science, Kings College London, Tech. Rep. TR-09-03, Apr. 2009.

[12] S. Elbaum, A. Malishevsky, and G. Rothermel, “Incorporating varying test costs and fault severities into test case prioritization,” in Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society, May 2001, pp. 329–338.

[13] S. Yoo, M. Harman, P. Tonella, and A. Susi, “Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge,” in Proceedings of the eighteenth international symposium on Software testing and analysis. ACM, Jul. 2009, pp. 201–212.

[14] D. Leon and A. Podgurski, “A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases,” in Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on. IEEE, Nov. 2003, pp. 442–453.

[15] E. G. Cartaxo, F. G. O. Neto, and P. D. Machado, “Automated test case selection based on a similarity function.” GI Jahrestagung (2), vol. 7, Sep. 2007, pp. 399–404.

[16] H. Hemmati, A. Arcuri, and L. Briand, “Achieving scalable model-based testing through test case diversity,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 22, no. 1, Feb. 2013, p. 6.

[17] D. Gusfield, Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press, 1997.

[18] R. A. Wagner and M. J. Fischer, “The string-to-string correction problem,” Journal of the ACM (JACM), vol. 21, no. 1, Jan. 1974, pp. 168–173.

[19] D. Kuemper, E. Reetz, and R. Tonjes, “Test derivation for semantically described iot services,” in Future Network and Mobile Summit (FutureNetworkSummit), 2013. IEEE, Jul. 2013, pp. 1–10.

[20] Testing Technologies, “TTworkbench,” Website, Aug 2014, available online at http://www.testingtech.com/products/ttworkbench.php retrieved: Aug. 2014.

Algebraic Analysis of Branching Processes

David Delfieu

Université de Nantes
IRCCyN
Nantes, France
david.delfieu@univ-nantes.fr

Maurice Comlan, Médésu Sogbohossou

Université d'Abomey-Calavi
LETIA
Abomey-Calavi, Bénin
comlan@hotmail.fr, sogbohossou_medesu@yahoo.fr

Abstract—**Combinatory explosion is a limit which can be encountered when a state space exploration is driven on large specification modeled with Petri nets. Technics like unfolding have been proposed to cope with this problem. This paper presents an axiomatic model to reduce unfoldings to canonic forms which preserves conflicts.**

Keywords–Petri Nets; Unfolding; Branching process; Algebra.

I. Introduction

Petri nets are a widely used tool used to model critical real-time systems. The formal verification of properties is then based on the computation of state space [1]. But, this computation faces generally, for highly concurrent and large systems, to combinatory explosion. A major cause is the semantics of interleaving. Partial order semantics [2] have been introduced to shunt those interleavings. This work, initiated in [3], go further with the introduction of the conflict equivalence. An operator which is an abstraction of sequence and true parallelism simplifies the representation of processes, only conflicts are preserved. This approach can be used to speed up the identification of the branching processes of an unfolding. The notion of equivalence can be used to make a new type of reduction of unfoldings.

Finite prefixes of net unfoldings constitute a first transformation of the initial Petri Net (PN), where cycles have been flattened. This computation produces a process set where conflicts act as a discriminating factor. A conflict partitions a process in branching processes. An unfolding can be transformed into a set of finite branching processes. Theses processes constitute a set of acyclic graphs - several graphs can be produced when the PN contains parallelism - built with events and conditions, and structured with two operators: causality and true parallelism. An interesting particularity of an unfolding is that in spite of the loss of the concept of global marking, these processes contain enough information to reconstitute the reachable markings of the original Petri nets. In most of the cases, unfoldings are larger than the original Petri net. This is provoked essentially when values of precondition places exceed the precondition of non simple conflicts. This produces a lot of alternative conditions. In spite of that, a step has been taken forward: cycles have been broken and the conflicts have structured the nets in branching processes.

A lot of works have been proposed to improve unfolding algorithms [2][4][5][6]. Is there another way to draw on recent works about unfolding? In spite of the eventual increase of the size of the net unfoldings, the suppression of conflicts and loops has decreased its structural complexity, allowing to compute the state space and to the extract of semantics.

From a developer's point of view an unfolding can be efficiently coded by a boolean table of events. This table describes every pair to pair relation between events. This table has been the starting point of our reflection: it stresses the point that a new connector can be defined to express that a set of events belong to the same process. This connector allows to aggregate all the events of a branching process. For example, a theorem is proposed to compute all the branching processes, in canonic form, for chains of conflicts of the kind illustrated in Figure 1.

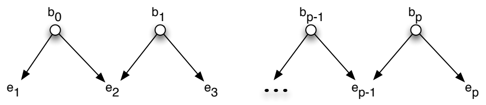


Fig 1. Chain of conflicts.

The work presented in this paper takes place in the context of combining process algebra [7][8] and Petri nets [9]. The axiomatic model of Milner's process with Calculus of Communicating Systems (CCS) is compared with the branching processes and related to other works in Section II. Then, after a brief presentation of Petri nets and unfoldings in Section III, Section IV presents our contribution with the definition of an axiomatic framework and the description of properties. The last section presents examples, in particular, illustrating a conflict equivalence.

II. Related Works

Process algebra appeared with Milner [8] on the Calculus of Communicating Systems (CCS) and the Communicating Sequential Processes (CSP) of Hoare [7] in not equivalent but similar approaches. The algebra of branching process we propose in this paper is inspired by the process algebra of Milner. *CCS* is based on two central ideas: The notion of observability and the concept of synchronized communication; *CCS* is as an abstract code that corresponds to a real program whose primitives are reduced to simple send and receive on channels. The terms (or agents) are called processes with interaction capabilities that match requests communication channels. The elements of the alphabet are observable events and concurrent systems (processes) can be specified with the use of three operators: sequence, choice, and parallelism. A main axiom of *CCS* is the rejection of distributivity of the sequence upon the choice. Let p and q be two processes, the complete process of syntax is:

$$\begin{array}{llll} \textit{Capacity} & \alpha & := & \bar{x} \mid x \mid \tau \\ \textit{Proces} & p & ::= & \alpha.p \mid p||q \mid p+q \mid D(\tilde{x}) \mid p\backslash x \mid 0 \end{array}$$

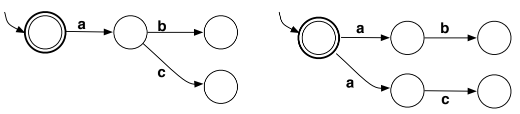


Fig 2. Milner: rejection of distributivity of sequence on choice.

Consider an observer. In the first automaton of the Figure 2, after the occurrence of the action a, he can observe either b or c. In the second automaton, the observation of a does not imply that b and c stay observable. The behavior of the two automata are not equivalent.

In *CCS*, Milner defines the observational equivalence. Two automata are observational equivalent if there are bisimular. On a algebraic point of view, the distributivity of the sequence on the choice is rejected in the equation (1):

$$a.(b+c) \not\equiv_{behaviorally} a.b + a.c \quad (1)$$

The key point of our approach is based on the fact that this distributivity is not rejected in occurrence nets. The timing of the choices in a process is essential [10]. The nodes of occurrence nets are events. An event is a fired transition of the underlying Petri net. In *CCS*, an observer observes possible futures. In occurrence nets, the observer observes arborescent past. This controversy in the theory of concurrency is an important topic of linear time versus branching time. In our model the equation (2) holds:

$$a \prec (b \perp c) \equiv (a \prec b) \perp (a \prec c) \quad (2)$$

The equation (2) is a basic axiom of our algebraic model. The equivalence relation differs then from bisimulation equivalence. This relation will be defined in the following with the definition of the canonic form of an unfolding.

Branching process does not fit with process algebra on numerous other aspects. For example, a difference can be noticed about parallelism. While unfolding keeps true parallelism, process algebra considers a parallelism of interleaving. Another difference is relative to events and conditions which are nodes of different nature in an unfolding. Conditions and events differ in term of ancestor. Every condition is produced by at most one event ancestor (none for the condition standing for m_0, the initial marking), whereas every event may have 1 or n condition ancestor(s). In *CCS*, there is no distinction between conditions and events. Moreover, conditions will be consumed defining processes as set of events.

However, a lot of works [6][10][11] have shown the interest of an algebraic formalization: it allows the study of connectives, the compositionally and facilitates reasoning (tools like [12]). Let have two Petri nets; it is questionable whether they are equivalent. In principle, they are equivalent if they are executed strictly in the same manner. This is obviously a too restrictive view they may have the same capabilities of interaction without having the same internal implementations. These work resulted to find matches (rather flexible and not strict) between nets. Mention may be made among other the occurrence net equivalence [13], the bisimulation equivalence [14], the partial order equivalence [15], or the ST-bisimulation equivalence [16]. These different equivalences are based either on the isomorphism between the unfolding of nets or on observable actions or traces of the execution of Petri nets or other criteria. This approach in this paper is weaker than a trace equivalence; it does not preserves traces but preserves conflicts. The originality of this approach is to encapsulate causality and concurrency in a new operator which "aggregates" and "abstract" events in a process. This new operator reduces the representation and accelerates the reduction process. This paper intends first, to give an algebraic model to an unfolding, and second, to establish a canonic form leading to the definition of an equivalence conflict.

III. Unfolding a Petri Net

A. Petri Net

A Petri net [9] $\mathcal{N} =< P,T,\mathcal{W} >$ is a triple with: P, a finite set of places, T, the finite set of transitions, $P \cup T$ are nodes of the net; $(P \cap T = \emptyset)$, and $\mathcal{W} :\ (P \times T)\ \cup\ (T \times P)\ \longrightarrow\ \mathcal{N}$, the flow relation defining arcs (and their valuations) between nodes of $\mathcal{N}$.

The pre-set (resp. post-set) of a node x is denoted ${}^\bullet x = \{y \in P \cup T \mid \mathcal{W}(y,x) > 0\}$ (resp. $x^\bullet = \{y \in P \cup T \mid \mathcal{W}(x,y) > 0\}$). A marking of a Petri net $\mathcal{N}$ is a mapping $m :\ P\ \longrightarrow\ \mathcal{N}$. A transition $t \in T$ is said *enabled* by m iff: $\forall p \in {}^\bullet t,\ m(p) \geq \mathcal{W}(p,t)$. This is denoted: $m \xrightarrow{t}$ Firing of t leads to the new marking m' $(m \xrightarrow{t} m')$: $\forall p \in P,\ m'(p) = m(p) - \mathcal{W}(p,t) + \mathcal{W}(t,p)$. The initial marking is denoted m_0.

A Petri net is *k-bounded* iff $\forall m$, reachable from $m_0, m(p) \leq k$ (with $p \in P$). It is said *safe* when *1-bounded*. Two transitions are in a *structural conflict* when they share at least one pre-set place; a conflict is *effective* when these transitions are both enabled by a same marking. The considered Petri nets in this paper are k-bounded.

B. Unfolding

In [4], the notion of *branching process* is defined as an initial part of a run of a Petri net respecting its partial order semantics and possibly including non deterministic choices (conflicts). This net is acyclic and the largest branching process of an initially marked Petri net is called *the* unfolding of this net. Resulting net from an unfolding is a labeled occurrence net, a Petri net whose places are called *conditions* (labeled with their corresponding place name in the original net) and transitions are called *events* (labeled with their corresponding transition name in the original net).

An occurrence net [17] $\mathcal{O} =< \mathcal{B},\mathcal{E},\mathcal{F} >$ is a 1-valued arcs Petri net, with $\mathcal{B}$ the set of conditions, $\mathcal{E}$ the set of events, and $\mathcal{F}$ the flow relation (1-valued arcs), such that: $|{}^\bullet b| \leq 1$ ($\forall b \in \mathcal{B}$), ${}^\bullet e \neq \emptyset$ $(\forall e \in \mathcal{E})$, and $\mathcal{F}^+$ (the transitive closure of $\mathcal{F}$) is a strict order relation. This net $\mathcal{O}$ is a set of acyclic graphs. $Min(\mathcal{O}) = \{b \mid b \in \mathcal{B}, |{}^\bullet b| = 0\}$ is the minimal conditions set: the set of conditions with no ancestor can be mapped with the initial marking of the underlying Petri net. Also, $Max(\mathcal{O}) = \{x \mid x \in \mathcal{B} \cup \mathcal{E}, |x^\bullet| = 0\}$ are maximal nodes.

Three kinds of relations could be defined between the nodes of $\mathcal{O}$:

- The strict causality relation noted $\prec$: $\forall x,y \in \mathcal{B} \cup$

$\mathscr{E}$, $x \prec y$ if $(x,y) \in \mathscr{F}^+$

- The conflict relation noted #: $\forall b \in \mathscr{B}$, if $e_1, e_2 \in b^\bullet$ $(e_1 \neq e_2)$, then e_1 and e_2 are in *conflict relation*, denoted e_1 # e_2 (in Figure 3.b $e_4 \# e_5$).
- The concurrency relation noted $\wr$: $\forall x,y \in \mathscr{B} \cup \mathscr{E}$ $(x \neq y)$, $x \wr y$ ssi $\neg((x \prec y) \vee (y \prec x) \vee (x \sharp y))$. (in Figure 3.b $e_2 \wr e_3$).

Remark 3.1: The transitive aspect of $\mathscr{F}^+$ implies a transitive definition of strict causality.

A set $B \subseteq \mathscr{B}$ of conditions such as $\forall b,b' \in B, b \neq b' \Rightarrow b \wr b'$ is a *cut*. Let B be a *cut* with $\forall b \in B, \nexists b' \in \mathscr{B} \backslash B,\ b \wr b'$, B is the *maximal cut*.

Definition 3.2: The unfolding $Unf_F \overset{\text{def}}{=} < \mathscr{O}_F, \lambda_F >$ of a marked net $< \mathscr{N}, m_0 >$, with $\mathscr{O}_F \overset{\text{def}}{=} < \mathscr{B}_F, \mathscr{E}_F, \mathscr{F}_F >$ an occurrence net and $\lambda_F :\ \mathscr{B}_F \cup \mathscr{E}_F \ \rightarrow\ \mathscr{P} \cup \mathscr{T}$ (such as $\lambda(\mathscr{B}_F) \subseteq \mathscr{P}$ and $\lambda(\mathscr{E}_F) \subseteq \mathscr{T}$) a labeling function, is given by:

1) $\forall p \in \mathscr{P}$, if $m_0(p) \neq \emptyset$, then $B_p \overset{\text{def}}{=} \{b \in \mathscr{B}_F \mid \lambda_F(b) = p \wedge {}^\bullet b = \emptyset\}$ and $m_0(p) = |B_p|$;
2) $\forall B_t \subseteq \mathscr{B}_F$ such as B_t is a cut, if $\exists t \in \mathscr{T}, \lambda_F(B_t) = {}^\bullet t \wedge |B_t| = |{}^\bullet t|$, then:
   a) $\exists ! e \in \mathscr{E}_F$ such as ${}^\bullet e = B_t \wedge \lambda_F(e) = t$;
   b) if $t^\bullet \neq \emptyset$, then $B'_t \overset{\text{def}}{=} \{b \in \mathscr{B}_F \mid {}^\bullet b = \{e\}\}$ is as $\lambda_F(B'_t) = t^\bullet \wedge |B'_t| = |t^\bullet|$;
   c) if $t^\bullet = \emptyset$, then $B'_t \overset{\text{def}}{=} \{b \in \mathscr{B}_F \mid {}^\bullet b = \{e\}\}$ is as $\lambda_F(B'_t) = \emptyset \wedge |B'_t| = 1$;
3) $\forall B_t \subseteq \mathscr{B}_F$, if B_t is not a cut , then $\nexists e \in \mathscr{E}_F$ such as ${}^\bullet e = B_t$.

The definition 3.2 represents an *exhaustive* unfolding algorithm of $< \mathscr{N}, m_0 >$. In 1., the algorithm for the building of the unfolding starts with the creation of conditions corresponding to the initial marking of $< \mathscr{N}, m_0 >$ and in 2., news events are added one at a time together with their output conditions (taking into account sink transitions). In 3., the algorithm requires that any event is a possible action: there are no adding nodes to those created in item 1 and 2. The algorithm does not necessary terminate; it terminates if and only if the net $< \mathscr{N}, m_0 >$ does not have any infinite sequence. The sink transitions (ie $t \in \mathscr{T}, t^\bullet = \emptyset$) are taken into account in 2.(c).

Let be $\mathscr{E} \subset \mathscr{E}_F$. The occurrence net $\mathscr{O} \overset{\text{def}}{=} < \mathscr{B}, \mathscr{E}, \mathscr{F} >$ associated with $\mathscr{E}$ such as $\mathscr{B} \overset{\text{def}}{=} \{b \in \mathscr{B}_F \mid \exists e \in \mathscr{E}, b \in {}^\bullet e \cup e^\bullet\}$ and $\mathscr{F} \overset{\text{def}}{=} \{(x,y) \in \mathscr{F}_F \mid x \in \mathscr{E} \vee y \in \mathscr{E}\}$ is *a prefix* of $\mathscr{O}_F$ if $Min(\mathscr{O}) = Min(\mathscr{O}_F)$. By extension, $Unf \overset{\text{def}}{=} < \mathscr{O}, \lambda >$ (with λ, the restriction of λ_F to $\mathscr{B} \cup \mathscr{E}$) is a prefix of unfolding Unf_F.

It should be noted that, according to the implementation, the names (the elements in the sets $\mathscr{E}$ and $\mathscr{B}$) given to nodes in the same unfolding can be different. A name can be independently chosen in an implementation using a tree formed by its causal predecessors and the name of the corresponding nodes in $\mathscr{N}$ [4].

Definition 3.3: A causal net $\mathscr{C}$ is an occurrence net $\mathscr{C} \overset{\text{def}}{=} < \mathscr{B}, \mathscr{E}, \mathscr{F} >$ such as:

1) $\forall e \in \mathscr{E} : e^\bullet \neq \emptyset \ \wedge\ {}^\bullet e \neq \emptyset$;
2) $\forall b \in \mathscr{B} : |b^\bullet| \leq 1 \ \wedge\ |{}^\bullet b| \leq 1$.

Definition 3.4: $\mathscr{P}_i = (\mathscr{C}_i, \lambda_F)$ is a *process* of $< \mathscr{N}, m_0 >$ iff: $\mathscr{C}_i \overset{\text{def}}{=} < \mathscr{B}_i, \mathscr{E}_i, \mathscr{F}_i >$ is a causal net and $\lambda :\ \mathscr{B}_i \cup \mathscr{E}_i\ \rightarrow P \cup T$

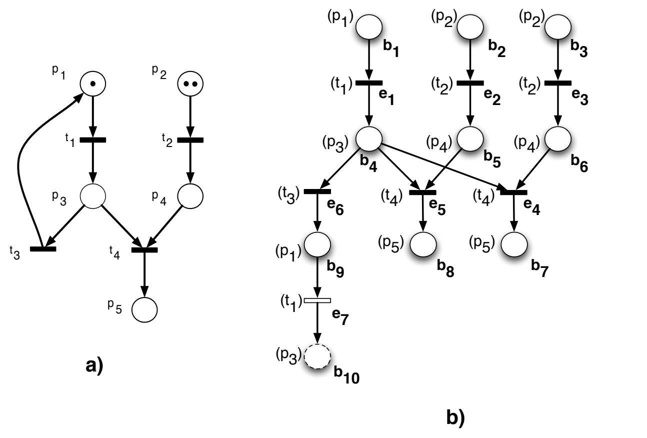


Fig 3. a) Petri net, b) Unfolding.

is a labeling fonction such as:

1) $\mathscr{B}_i \subseteq \mathscr{B}_F$ *and* $\mathscr{E}_i \subseteq \mathscr{E}_F$
2) $\lambda_F(\mathscr{B}_i) \subseteq P$ *and* $\lambda_F(\mathscr{E}_i) \subseteq T$;
3) $\lambda_F({}^\bullet e) = {}^\bullet \lambda_F(e)$ *and* $\lambda_F(e^\bullet) = \lambda_F(e)^\bullet$
4) $\forall e_i \in \mathscr{E}_i,\ \ \forall p \in P :\ \ \mathscr{W}(p, \lambda_F(e)) = |\lambda^{-1}(p) \cap {}^\bullet e| \ \wedge\ \mathscr{W}(\lambda_F(e), p) = |\lambda^{-1}(p) \cap e^\bullet|$
5) If $p \in Min(P) \Rightarrow \exists b \in \mathscr{B}_i :\ {}^\bullet b = \emptyset\ \wedge\ \lambda_F(b) = p$

$Max(\mathscr{C}_i)$ is the *state* of $\mathscr{N}$. $Min(\mathscr{C}_i)$ and $Max(\mathscr{C}_i)$ are maximum cuts. Generally, any maximal cut $B \subseteq \mathscr{B}_i$ corresponds to a reachable marking m of $< \mathscr{N}, m_0 >$ such as $\forall p \in \mathscr{P}, m(p) = |B_p|$ avec $B_p = \{b \in B \mid \lambda(b) = p\}$.

The *local configuration* of an event e is defined by: $[e] \overset{\text{def}}{=} \{e' \mid e' \prec e\} \cup \{e\}$ and is a process. For example of unfolding in Figure 3.b: $[e_4] \overset{\text{def}}{=} \{e_1, e_3, e_4\}$.

The conflits in a unfolding derive from the fact that there is an reachable marking (a cut in an unfolding) such as two or many transitions of a labelled net $< \mathscr{N}, m_0 >$ are *enabled* and the firing of one transition disable other. Whence the proposition:

Proposition 3.5: Let be $e_1, e_2 \in \mathscr{E}_F$. If $e_1\ \perp\ e_2$, then there $\exists (e'_1, e'_2) \in [e_1] \times [e_2]$ such as ${}^\bullet e'_1 \cap {}^\bullet e'_2 \neq \emptyset$ et ${}^\bullet e'_1 \cup {}^\bullet e'_2$ is a cut.

IV. BRANCHING PROCESS ALGEBRA

The previous section showed how unfolding exhibits causal nets and conflicts. Otherwise, every couple of events which are not bounded by a causal relation or the same conflict set are in concurrency. Then, an unfolding allows to build a 2*D*-table making explicit every binary relations between events. Practically, this table establishes the relations of causality and exclusion. If a binary relation is not explicit in the table, it means that the couple of events are in a concurrency relation.

Let $\mathscr{EB} = \mathscr{E} \cup \mathscr{B}$ a finite alphabet, composed of the events and the conditions generated by the unfolding. The event table (produced by the unfolding) defines for every couple in $\mathscr{EB}$ either a causality relation $\mathscr{C}$, either a concurrency relation $\mathscr{I}$ or an exclusive relation $\mathscr{X}$. These sets of binary relations dot not intersect and the following expressions can be deduced:

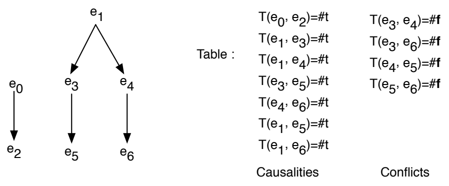
$$Unf/_{\mathscr{X}} = \mathscr{C} \cup \mathscr{I} \tag{3}$$
$$Unf/_{\mathscr{C}} = \mathscr{X} \cup \mathscr{I} \tag{4}$$
$$Unf/_{\mathscr{I}} = \mathscr{C} \cup \mathscr{X} \tag{5}$$

To illustrate these relation sets, the negation operator noted $\neg$ can be introduced. Then, the equations (3),(4),(5) leads to (6),(7),(8):

$$\neg((e_1,e_2) \in \mathscr{I}) \;\Leftrightarrow\; (e_1,e_2) \in \mathscr{C} \cup \mathscr{X} \tag{6}$$
$$\neg((e_1,e_2) \in \mathscr{C}) \;\Leftrightarrow\; (e_1,e_2) \in \mathscr{I} \cup \mathscr{X} \tag{7}$$
$$\neg((e_1,e_2) \in \mathscr{X}) \;\Leftrightarrow\; (e_1,e_2) \in \mathscr{C} \cup \mathscr{I} \tag{8}$$

The equation (8) expresses that if two events are not in conflict they are in the same branching process. Let us now define the union of binary relations $\mathscr{C}$ and $\mathscr{I}$: $\mathscr{P} = \mathscr{C} \cup \mathscr{I}$ For every couple $(e_1,e_2) \in \mathscr{P}$, either (e_1,e_2) are in causality or in concurrency: $\mathscr{P}$ is the union of every branching process of an unfolding.

a) Example: Let us consider an unfolding (left part) on the Figure 4 and the table T (right part) which is its representation: In Figure 4, the table T contains 7 causal



Causalities	Conflicts
T(e0, e2)=#t	T(e3, e4)=#f
T(e1, e3)=#t	T(e3, e6)=#f
T(e1, e4)=#t	T(e4, e5)=#f
T(e3, e5)=#t	T(e5, e6)=#f
T(e4, e6)=#t	
T(e1, e5)=#t	
T(e1, e6)=#t	

Fig 4. Unfolding.

relations and 4 conflict relations. (e_0,e_4) is not (negation) in the table, expresses that e_0 and e_4 are concurrent. Moreover, if two events are not in conflict (consider e_0 and e_6): (e_0,e_6) is not a key of the table, (e_0,e_6) are in concurrency and thus, those events belongs to the same branching process.

A. Definition of the Algebra

The starting point of this work is based on the fact that the logical negation operator articulates the relation between two sets: the process set $\mathscr{P}$, and the exclusion set $\mathscr{X}$. As mentioned in Section IV, $\mathscr{C}$, $\mathscr{I}$ *and* , $\mathscr{P}$ does not intersect, then semantically, if a couple of events is not in a relation of exclusion (noted $\perp$), the events are in $\mathscr{P}$. $\mathscr{P}$ contains binary relations between events that are in branching process.

To express that events are in the same branching process, a new operator noted $\oplus$ is introduced. An algebra describing branching process can be defined as follow:

$$\{\mathscr{U}, \prec\,,\, \wr\,,\, \perp\,, \oplus,\, \neg\}$$

Let us note; $* = \oplus, \prec$, or $\perp$, $\#t$ the void process, and $\#f$ the false process. Here is the formal signature of the language:

- $\forall e \in \mathscr{E}\mathscr{B}, e \in \mathscr{U}, \#t \in \mathscr{U}, \#f \in \mathscr{U}$
- $\forall e \in \mathscr{U}, \neg e \in \mathscr{U}$
- $\forall (e_1,e_2) \in \mathscr{U}^2, e_1 * e_2 \in \mathscr{U}$

Properties, neutral/absorbing elements, distributivities and semi-distributivities have been defined in [3]. However, let us now just recall the definitions (equations (9),(10),(11),(12)):

1) Causality: $\mathscr{C}$ is the set of all the causalities between every elements of $\mathscr{E}\mathscr{B}$. $e_1 \prec e_2$ if e_1 is in the local configuration of e_2:

$$e_1 \prec e_2 \text{ if } e_1 \in [e_2] \tag{9}$$

2) Exclusion: $\mathscr{X}$ is the set of all the exclusion relations between every elements of $\mathscr{E}\mathscr{B}$. Two events are in exclusion *iff* they are either in direct conflict, either it exists a conflict at any level with an ancestor:

$$e_1 \perp e_2 \equiv ((\bullet e_1 \cap \bullet e_2 \neq \emptyset) \text{ or } (\exists e_i, e_i \prec e_2 \text{ and } e_1 \perp e_i)) \tag{10}$$

3) Concurrency: $\mathscr{I}$ is the set of every couple of element of $\mathscr{E}\mathscr{B}$ in concurrency. e_1 and e_2 are in concurrency if the occurrence of one is independent of the occurrence of the other. So, $e_1 \wr e_2$ iff e_1 and e_2 are neither in causality neither in exclusion.

$$e_1 \wr e_2 \equiv \neg((e_1 \perp e_2) \text{ or } (e_1 \prec e_2) \text{ or } (e_2 \prec e_1)) \tag{11}$$

4) Process: $\oplus$ aggregates events in one process. Two events e_1 and e_2 are in the same process if e_1 causes e_2 or if e_1 is concurrent with e_2:

$$e_1 \oplus e_2 \equiv (e_1 \prec e_2) \text{ or } (e_2 \prec e_1) \text{ or } (e_1 \wr e_2) \tag{12}$$

This operator constitutes an abstraction which hides in a black box causalities and concurrency. The meaning of this operator is similar to the linear connector $\oplus$ of MILL [18]. It allows to aggregates resources. But, in the context of unfolding, events or conditions are unique and then they cannot be counted. Thus, this operator is here idempotent.

The expression $e_1 \oplus e_2$ defines that e_1 and e_2 are in the same process.

Note that $(\oplus\ e_1\ e_2\ ...\ e_{n-1}\ e_n)$ will abbreviate $(e_1 \oplus\ e_2 \oplus e_3 \oplus ... e_{n-1} \oplus\ e_n)$

B. Axioms

The following axioms stem directly from previous assumptions and definitions made upon the algebraic model:

Axiom 4.1 (Distributivity of $\prec$):

$$e \prec (e_1 \perp e_2) \equiv_{def} (e \prec e_1) \perp (e \prec e_2)$$

The first axiom constitutes the basis of our approach. As discussed in the Section II, on the contrary of *CCS*, e is distributed onto two expressions, giving alternative process.

Axiom 4.2 (Definition of $\oplus$):

$$e_1 \oplus e_2 \equiv_{def} (e_1 \prec e_2) \perp (e_2 \prec e_1) \perp (e_1 \wr e_2)$$

$\oplus$ aggregates two elements in a process. Two elements are in a process if they are concurrent or in a causality relation.

Axiom 4.3 ($\prec$):

$$e_1 \prec e_2 \equiv_{def} \neg e_1 \perp (e_1 \oplus e_2)$$

A causality can be expressed by two processes in exclusion: either $\neg e_1$: e_1 has not occurred either $e_1 \oplus e_2$: e_1 and e_2 within the same process.

Axiom 4.4 (Duality between $\oplus$ and $\perp$):

$$e_1 \oplus e_2 \equiv_{def} e_1 \neg\!\perp e_2 \qquad e_1 \neg\!\oplus e_2 \equiv_{def} e_1 \perp e_2$$

This axiom comes from the introduction of the operator $\neg$ discussed in the beginning of the Section IV. It expresses that $\mathscr{P}$ and $\mathscr{X}$ are complementary sets.

Axiom 4.5 (Exclusion):

$$e_1 \perp e_2 \equiv_{def} (\neg e_1 \oplus e_2) \perp (e_1 \oplus \neg e_2)$$

The fifth axiom expresses that a conflict can be considered as two processes in conflict

Axiom 4.6 (Distributivities):

- $\prec, \perp, \oplus$ are distributive over $\wr$.
- $\prec, \oplus, \wr$ are distributive over $\perp$ (axiom 4.3).
- $\perp, \wr$ are distributive over $\perp$ and $\oplus$.

The distributivities over $\perp$ are used in the transformation of an expression in the canonic form. The other distributivities will be used in the reduction process.

C. Canonic Form

1) Definition: The definition of the canonic form allows to define an equivalence called a "conflict equivalence".

Definition 4.1: A canonic process is a formula expressed on elements of $\mathscr{E}\mathscr{B}$ and with the operators $\oplus$, $\perp$ ordered by an alphanumeric sort on the name of its symbol.

Theorem 4.2 (Canonical form): Let us consider an unfolding U, this form can be reduced in the following form:

$$U = \ (\perp \ P_1 \ P_2 \ ... \ P_n) \qquad \text{where} \quad P_i = (\oplus \ e_{i_1} ... \ e_{i_n})$$

This form is canonic and exhibits every processes P_i of the unfolding.

Proof: In an unfolding every causality ($\prec$) and every partial order ($\wr$) can be reduced in $\oplus$ by deduction rules Modus Ponens (MP,MP_1,MP_2), Simplification rule (S) and *Par* (see Section IV-C2). Moreover, $\oplus$ and $\perp$ are mutually distributive, so $\perp$ can be factorized in every sub-formula to reach the higher level of the formula. In fine, an alphanumeric sort on symbols of the processes can be applied to assure the unicity of the form. ∎

This canonic form preserves conflicts, let us now define a *conflict equivalence*:

Definition 4.3 (Conflict Equivalence): Let us U_1, U_2 unfoldings of Petri nets:

$$U_1 \approx_{conf} U_2 \qquad \text{iff they have the same canonic form.}$$

Remark 4.4: A process is an aggregate set of events where $\prec$ and $\wr$ are hidden. This equivalence is lower than a trace equivalence: each process P_i is an abstraction of a set of traces.

2) Derivation Rules: This section gives a set of rules which transform branching processes toward a canonical form. Theses transformations preserve conflicts whereas $\prec$ and $\wr$ are transformed in $\oplus$.

Let us note b a condition, e an event and E a well formed formula on the algebra. Theses rules allow to reduct process:

1) Modus Ponens:

$$\frac{\vdash \oplus\ b... \quad \vdash \oplus\ b... \prec e}{\vdash e}\ MP_1 \qquad\qquad \frac{\vdash e \quad \vdash e \prec \oplus\ b...}{\vdash \oplus\ e\ b...}\ MP_2$$

Where $\oplus b...$ stands for the general form for $\oplus b_1 b_2 ...\ b_n$. MP_1 expresses that $b...$ are consumed by the causality, whereas, in MP_2 e stays in the conclusion.

2) Dual form: $$\frac{\vdash \neg e_1 \quad \vdash e_1 \prec e_2}{\vdash \neg e_1 \oplus \neg e_2}\ MP'$$

3) Simplification: $$\frac{\vdash \neg e_1 \oplus E}{\vdash E}\ S_1 \qquad\qquad \frac{\vdash \oplus b...\ E}{\vdash E}\ S_2$$

Those rules are applied, *in fine*, to clear not pertinent informations in the process. S_1 rule is applied, to clear the negations whereas S_2 is applied to clear the conditions which have not been consumed.

4) Reduction of $\wr$: $$\frac{\vdash e_1 \wr e_2}{\vdash e_1 \oplus e_2}\ Par$$

This rule corresponds to the definition of $\oplus$
These rules have been defined to lead to a canonic form.

D. Theorems

The properties of operators (definition, axioms and distributivites) allow to define theorems which are congruences w.r.t the operators of Section IV-A (proofs have been already stated in [3]).

Theorem 4.5 (Conflict):

$$e_1 \prec (e_2 \perp e_3) \equiv (e_1 \prec (e_2 \oplus \neg e_3)) \perp (e_1 \prec (\neg e_2 \oplus e_3))$$

This theorem expresses how to develop a conflict and the following theorem allows to reduce processes:

Theorem 4.6 (Absorption): Let E, F some processes: $E \perp (E \oplus F) \equiv E \oplus F$

1) Chain of conflicts: This section presents a theorem which computes the branching process in canonic form of a chain of conflict illustrated in Figure 5.

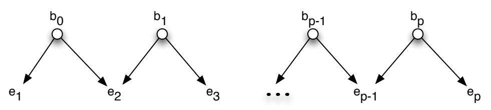


Fig 5. Chain of conflicts.

The axiomatic representation of the unfolding is:

$$U = ((\oplus\ b_0\ b_1\ ...\ (b_0 \prec (e_1 \perp e_2))(b_1 \prec (e_2 \perp e_3))...)$$

After some steps of reduction $(MP+S)$:

$$U = (e_1 \perp e_2 \perp ... \perp\ e_p)$$

Let us now consider the following conventions: Let us note:

- $l^1 = (e_1, e_2, ...e_n)$, $l^2 = (e_2, ...e_n)$
- l_i the i^{th} element of a list l.
- If e_i is an element of the list l, let us note $indice(e_i)$ the position of e_i in l.

Remark 4.7: In the list of event constituting a chain of conflict ($l = (e_1, e_2, ...e_n)$), for every event e_i, the next (resp. previous) event in the same branching process is e_{i+2} or e_{i+3} (resp. e_{i-2} or e_{i-3})

The next definition defines two processes U_n and V_n which are aggregation of events where the possible successor of an event e_i is either $l_{(indice(e_i)+2)}$ either $l_{(indice(e_i)+3)}$.

Definition 4.8: Let us consider that $n <= p$ (p: index of the last event implied in the chain of conflict):

$$\begin{cases} U_0 &= e_1 \\ U_n^1 &= l_{n+2}^1 \oplus U_{n+2}^2 \\ U_n^2 &= l_{n+3}^1 \oplus U_{n+3}^2 \\ U_n &= U_n^1 \quad \oplus U_n^2 \end{cases} \qquad \begin{cases} V_0 &= e_2 \\ V_n^1 &= l_{n+2}^2 \oplus V_{n+2}^2 \\ V_n^2 &= l_{n+3}^2 \oplus V_{n+3}^2 \\ V_n &= V_n^1 \quad \oplus V_n^2 \end{cases}$$

U_n: processes beginning by e_1 V_n: processes beginning by e_2

Theorem 4.9: The canonic form of a chain of conflict C is $U_n \oplus V_n$:

$$(e_1 \perp \ e_2 \perp \ ... \perp \ e_p) \equiv U_n \oplus V_n$$

V. EXAMPLES

A. Example 1

The Figure 6 gives a Petri net which represents a chain of conflicts and its unfolding. The unfolding gives a table of binary relations on events (see Section IV) which is represented by the following algebraic expression U_2:

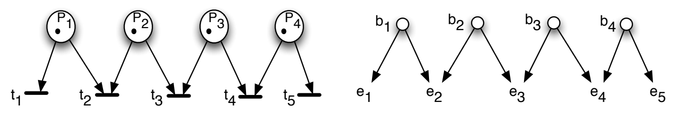


Fig 6. PN and unfolding of a chain of conflicts.

$$U_1 = (\oplus\ b_1\ b_2\ b_3\ b_4\ b_5\ (b_1 \prec (e_1 \perp e_2))\ (b_2 \prec (e_2 \perp e_3))\ ...)$$

After some steps of reduction ($MP + S$), U_1 becomes:

$$(e_1 \perp \ e_2 \perp \ e_3 \perp \ e_4 \perp \ e_5) \tag{13}$$

The theorem (T 4.9) allows to compute from (13) its following canonic form:

$$(\perp (\oplus\ e_1\ e_3\ e_5)(\oplus\ e_1\ e_4\ )(\oplus\ e_2\ e_4)(\oplus\ e_2\ e_5))$$

B. Example 2

Let us consider the following Unfolding of the Figure 7. The table computed which leads to the following algebraic expression U_2:

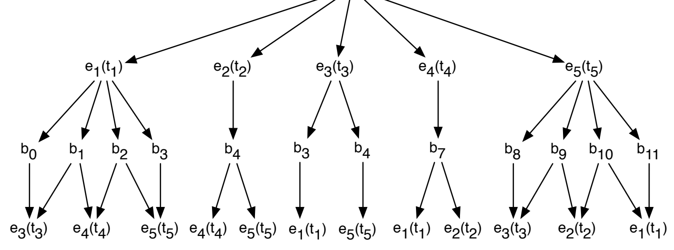


Fig 7. U_2.

$$\begin{aligned} U_2 \quad = (\oplus\ b_{12} \quad & (b_{12} \prec (e_1 \perp \ e_2 \perp \ e_3 \perp \ e_4 \perp \ e_5)) \\ & (e_1 \prec (\oplus\ b_0\ b_1\ b_2\ b_3))(e_2 \prec b_4)(e_3 \prec (\oplus b_5\ b_6))(e_4 \prec b_7) \\ & (e_5 \prec (\oplus b_8\ b_9\ b_{10}\ b_{11}))((\oplus\ b_0\ b_1) \prec e_3)\ ((\oplus\ b_1\ b_2) \prec e_4) \\ & ((\oplus\ b_2\ b_3) \prec e_5)\ (b_4 \prec (\perp e_4\ e_5))(b_5 \prec e_1)\ (b_6 \prec e_5) \\ & (b_7 \prec (\perp e_1\ e_2))\ ((\oplus\ b_8\ b_9) \prec e_3)\ ((\oplus\ b_9\ b_{10}) \prec e_2) \\ & ((\oplus\ b_{10}\ b_{11}) \prec e_1)) \end{aligned} \tag{14}$$

Let us note P the aggregation of the five first lines of the equation. (14) becomes:

$$U_2 \quad = (\oplus\ b_{12} \quad (b_{12} \prec (\perp\ e_1\ e_2\ e_3\ e_4\ e_5))\ P \tag{15}$$

Rules $MP1$, MP_2 and theorem 1, reduce (15) in:

$$U_2 \quad = (\perp (\oplus\ e_1\ P)\ (\oplus\ e_2\ P)\ (\oplus\ e_3\ P)\ (\oplus\ e_4\ P)\ (\oplus\ e_5\ P)\ )$$

Distributivity of *perp*:

$$\begin{aligned} U_2 \quad = (\oplus \quad & (\perp (\oplus\ e_1\ b_0\ b_1\ b_2\ b_3)(\oplus\ e_2\ b_4)(\oplus\ e_3\ b_5\ b_6)\ (\oplus\ e_4\ b_7) \\ & (\oplus\ e_5\ b_8\ b_9\ b_{10}\ b_{11}))\ ((\oplus\ b_0\ b_1) \prec e_3)\ ((\oplus\ b_1\ b_2) \prec e_4) \\ & ((\oplus\ b_2\ b_3) \prec e_5)\ (b_4 \prec (\perp e_4\ e_5))(b_5 \prec e_1)\ (b_6 \prec e_5) \\ & (b_7 \prec (\perp e_1\ e_2))((\oplus\ b_8\ b_9) \prec e_3)\ ((\oplus\ b_9\ b_{10}) \prec e_2) \\ & ((\oplus\ b_{10}\ b_{11}) \prec e_1)) \end{aligned}$$

Distributivity of $\perp$ and MP_1:

$$\begin{aligned} U_2 \quad = (\perp \quad & (\oplus\ e_1\ e_3\ e_5\ b_1\ b_2)(\oplus\ e_1\ e_4\ b_0\ b_3)(\oplus\ e_2\ e_4)(\oplus\ e_2\ e_5) \\ & (\oplus\ e_3\ e_1)(\oplus\ e_3\ e_5)(\oplus\ e_4\ e_1)(\oplus\ e_4\ e_2)(\oplus\ e_5\ e_1\ e_3\ b_9\ b_{10}) \\ & (\oplus\ e_5\ e_2\ b_8\ b_{11})) \end{aligned}$$

Theorem 2 (absorption of $(\oplus\ e_3\ e_1)$ and $(\oplus\ e_3\ e_5)$ in $(\oplus\ e_1\ e_3\ e_5\ b_1\ b_2)$, idempotency of $\perp$:

$$\begin{aligned} U_2 \quad = (\perp \quad & (\oplus\ e_1\ e_3\ e_5\ b_1\ b_2)(\oplus\ e_1\ e_4\ b_0\ b_3)(\oplus\ e_2\ e_4)(\oplus\ e_2\ e_5) \\ & (\oplus\ e_4\ e_1)(\oplus\ e_5\ e_1\ e_3\ b_9\ b_{10})(\oplus\ e_5\ e_2\ b_8\ b_{11})) \end{aligned}$$

Rules of simplification S_1 and S_2 and theorem 2:

$$U_2 = (\perp (\oplus\ e_1\ e_3\ e_5)(\oplus\ e_1\ e_4)(\oplus\ e_2\ e_4)(\oplus\ e_2\ e_5))$$

The two unfoldings of example 1 and 2 have the same canonic form, they are *conflict-equivalent*: $U_1 \approx_{conf} U_2$

1) Reasoning about processes: Let us consider all the process p of $U_2 : (\oplus\ e_1\ e_3\ e_5), (\oplus\ e_1\ e_4), ...$

- $\forall p \in U_2$ whenever e_3 is present, e_1 is present.
- $\forall p \in U_2, \neg e_3 \perp (e_1 \oplus\ e_3 \oplus e_5)$
  This is the algebraic definition of $\prec$. Finally from this chain of conflicts, the following causality can be deduced:
  $$e_3 \prec (e_1 \oplus e_5) \tag{16}$$
- A similar reasoning can be made:
  $$\forall p \in U_2, \neg(e_1 \oplus e_5) \perp (e_1 \oplus\ e_3 \oplus e_5)$$
  This is the algebraic definition of:
  $$(e_1 \oplus e_5) \prec e_3 \tag{17}$$
  (16) and (17) expresses that there is a *strong link* between e_3 and the process $(e_1 \oplus e_5)$ but $\prec$ is no well suited to encompass this relation. Theses two processes are like "intricated".

- In the same manner:

$$\begin{aligned} \neg e_2 \perp (e_2 \oplus e_4) \perp (e_2 \oplus e_5) &\equiv_{dist} \neg e_2 \perp (e_2 \oplus (e_4 \perp e_5)) \\ &\equiv_{def} e_2 \prec (e_4 \perp e_5) \qquad (18) \\ e_2 \text{ leads to a conflict} & \\ \neg e_1 \perp ((\oplus e_1 e_3 e_5) \perp (e_1 \oplus e_4) &\equiv_{dist} \neg e_1 \perp (e_1 \oplus ((e_3 \oplus e_5) \perp e_4)) \\ &\equiv_{def} e_1 \prec ((e_3 \ \oplus \ e_5) \perp e_4) \qquad (19) \end{aligned}$$

Equations (18) and (19) show that e_1 and e_2 transform the chain of conflict in a unique conflict. New relations between events or processes can be introduced:

- Alliance relation: e_1, e_3 and e_5 are in “an alliance relation”. Every event of this set is enforced by the occurrence of the other events: $e_1 \oplus e_3$ enforces e_5, $e_1 \oplus e_5$ enforces e_3 and $e_3 \oplus e_5$ enforces e_1.
- Intrication: the occurrence of e_3 forces $e_1 \oplus e_5$ and reciprocally $e_1 \oplus e_5$ forces e_3.
- Resolving conflicts (liberation):
  - e_1 resolves 3 conflicts on 4 (as e_2, e_4 and e_5)
  - e_3 resolves every conflicts.

Semantically, e_3 can be identified as an important event in the chain. Moreover $(\oplus e_1 \ e_3 \ e_5)$ is a process aggregated with “associated events”. This chain of conflict can be seen as two causalities in conflicts: $(e_1 \prec (e_4 \perp (e_3 \oplus e_5))) \ \perp \ (e_2 \prec (e_4 \perp e_5))$

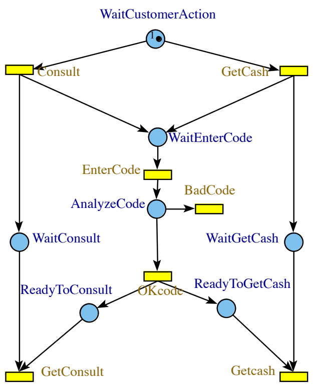


Fig 8. *Cash dispenser.*

2) Example 3 (Cash dispenser): Let us consider the cash dispenser of the Figure 8, its unfolding in canonical form is:

$U3 = (\perp (\oplus \ Consult \ EnterCode \ OKcode \ GetConsult)$
$(\oplus \ Consult \ EnterCode \ OKcode \ Getcash)$
$(\oplus \ Consult \ EnterCode \ BadCode)$
$(\oplus \ GetCash \ EnterCode \ OKcode \ Getcash)$
$(\oplus \ GetCash \ EnterCode \ OKcode \ GetConsult)$
$(\oplus \ GetCash \ EnterCode \ BadCode))$

This expression enlightens that *GetCash* and *BadCode* are neither in the same process.

VI. Conclusion and future work

This work is a first attempt to present an axiomatic framework to the analyze of the processes issued of an unfolding. From a set of axioms, distributivities and derivation rules, theorems have been established and a reduction process can lead to a canonic form The unfolding process, definitions, theorems and reduction rules have been coded in LISP[19] with a package named PLT/Redex[3][12]. This canonic form assets an equivalence conflicts ($\equiv_{conf}$) between unfoldings and then Petri nets.

Several perspectives are into progress. First, news theorems have to be established allowing to speed up the procedure of canonic reduction and to extend extraction of knowledge on relationship between events. Different kinds of relationship between events have to be defined and formalized: Alliance relation, Intrication, etc. Moreover, as already outlined in the last part of the example section, algebraic reasoning can raise semantic informations about events from the canonic form. Another perspective is to extend this approach to Petri nets with inhibitor and drain arcs.

References

[1] B. Berthomieu and F. Vernadat, “State class constructions for branching analysis of time petri nets,” in TACAS, volume 2619 of LNCS. Springer Verlag, 2003, pp. 442–457.

[2] J. Esparza and K. Heljanko, “Unfoldings - a partial-order approach to model checking,” EATCS Monographs in Theoretical Computer Science, 2008.

[3] D. Delfieu and M. Sogbohossou, “An algebra for branching processes,” in Control, Decision and Information Technologies (CoDIT), 2013 International Conference on, May 2013, pp. 625–634.

[4] J. Engelfriet, “Branching processes of petri nets,” Acta Informatica, vol. 28, no. 6, 1991, pp. 575–591.

[5] J. Esparza, S. Römer, and W. Vogler, An Improvement of McMillan’s Unfolding Algorithm. Mit Press, 1996.

[6] McMillan and Kenneth, “Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits,” in Computer Aided Verification. Springer, 1993, pp. 164–177.

[7] C. A. R. Hoare, Communicating sequential processes. Prentice-hall Englewood Cliffs, 1985, vol. 178.

[8] R. Milner, Communication and concurrency. Prentice-hall Englewood Cliffs, 1989.

[9] C. A. Petri, “Communication with automata,” PhD thesis, Institut fuer Instrumentelle Mathematik, 1962.

[10] R. Glabbeek and F. Vaandrager, “Petri net models for algebraic theories of concurrency,” in PARLE Parallel Architectures and Languages Europe, ser. Lecture Notes in Computer Science, J. Bakker, A. Nijman, and P. Treleaven, Eds. Springer Berlin Heidelberg, 1987, vol. 259, pp. 224–242.

[11] E. Best, R. Devillers, and M. Koutny, “The box algebra=petri nets+process expressions,” Information and Computation, vol. 178, no. 1, 2002, pp. 44 – 100.

[12] M. Felleisen, R. Findler, and M. Flatt, Semantics Engineering With PLT Redex. Mit Press, 2009.

[13] M. Nielsen, G. Plotkin, and G. Winskel, “Petri nets, event structures and domains,” in T. Theor. Comp. Sci., vol. 13(1), 1981, pp. 89–118.

[14] J. Baeten, J. Bergstra, and J. Klop, “An operational semantics for process algebra,” in CWI Report CSR8522, 1985.

[15] G. Boudol and I. Castellani, “On the semantics of concurrency: partial orders and transitions systems,” in Rapports de Recherche No 550, INRIA, Centre Sophia Antipolis, 1986.

[16] V. Glaabeek and F. Vaandrager, “Petri nets for algebraic theories of concurrency,” in CWI Report SC-R87, 1987.

[17] T. Chatain and C. Jard, “Complete finite prefixes of symbolic unfoldings of safe time petri nets,” in Petri Nets and Other Models of Concurrency - ICATPN 2006, ser. Lecture Notes in Computer Science, S. Donatelli and P. Thiagarajan, Eds. Springer Berlin Heidelberg, 2006, vol. 4024, pp. 125–145. [Online]. Available: http://dx.doi.org/10.1007/11767589

[18] J.-Y. Girard, “Linear logic,” Theoretical computer science, vol. 50, no. 1, 1987, pp. 1–101.

[19] G. L. Steele, Common LISP: the language. Digital press, 1990.

From Semantic IoT-Service Descriptions to Executable Test Cases - Information Flow of an Implemented Test Framework

Daniel Kuemper, Eike Reetz,
Marten Fischer and Ralf Toenjes

Lab for RF-Technology and Mobile Communications
University of Applied Sciences Osnabrueck
Osnabrueck, Germany
Email: {d.kuemper,e.reetz,m.fischer,r.toenjes}
@hs-osnabrueck.de

Elke Pulvermueller

Institute of Computer Science
University of Osnabrueck
Osnabrueck, Germany
Email: elke.pulvermueller@uni-osnabrueck.de

***Abstract*—Automated test derivation is expected to be one of the key drivers of a rapid creation of robust Internet of Things (IoT) applications. The paper describes a two-step approach how concepts for semantically described IoT services can be used to derive functional test cases to test services in a sandbox environment. In the first step, the description of the service is used to generate a state based model of the service behaviour and its interfaces. Therefore, a methodology to enrich service descriptions for (semi-) automated test derivation and the required IoT specific adaptations are discussed in detail. These descriptions are used to generate customised test data and to achieve full parameter combination coverage. In the second step, the generated extended finite state machine model is analysed to create test cases in a standardised testing notation. Utilising this two-step automation approach enables test developers to evaluate and influence resulting test cases. The implementation proves that the envisaged extension can amplify the usefulness of web services descriptions for the test derivation for IoT services by reducing the effort to create and execute test cases.**

Keywords–IoT; Model Based Testing; Test Derivation; Semantic Annotation; RESTful; TTCN-3; WADL.

I. INTRODUCTION

Distributed IoT services are becoming increasingly complex since the usage of sensors and actuators with atomic functionalities brings along a high variety of heterogeneous interfaces [1]. Therefore, it is crucial to employ functional tests to evaluate faultless service interaction. Manual test creation causes a high effort in analysing interfaces and service behaviour to find suitable test cases. This effort can be reduced by employing model-based test approaches [2]. This work illustrates how a two-step model-driven testing approach, which utilises explicit information representation at different abstraction levels, can be used to create test cases for semantically described Representational State Transfer based (RESTful) IoT services whilst regarding their stateful behaviour. A fully automated model based testing approach needs very extensive Input, Output, Precondition and Effect (IOPE) descriptions [3] and deprives the control of the test developer to evaluate tests dependent on the services usage. Therefore, this work tries to lower the effort for the service description by enabling a use case-based sequence description approach. Furthermore, it aims at reducing the effort for manually enhancing service descriptions compared to a full manual test case creation.

The remainder of the paper is structured as follows: After the discussion of the current state of the art in Section II, the overall project concept and implemented architecture is outlined in Section III. Detailed descriptions of the utilised annotation methods are shown in Section IV. Section V depicts an IoT example service, which is used in Section VI to discuss the test derivation process and its model transformation. The conclusion and outlook section completes the paper.

II. RELATED WORK

In recent years, a lot of research efforts have been invested in providing efficient ways to automate the process of testing software. Different strategies have been developed in generating test cases and providing them with adequate test data. Basically, three approaches can be identified. First, finding suspicious code through code analysis [4]. This approach requires access to the source code of the System Under Test (SUT) and is able to find unreachable code and other violations to coding rules. The second approach tries to find implementation faults by exploiting public interfaces with a large number of randomly generated data [5]. This fuzzing approach can find security relevant implementation errors (e.g., buffer overflow) but on the other hand it produces a very large number of test cases of rather poor quality. The employment of Equivalence Class Partitioning (ECP) can reduce the amount of test data and test cases by defining valid and invalid arguments for the interface invocation [6]. By employing more precise semantic service descriptions, the approach proposed in this work tries to overcome solely random generations by taking reusable parameter range definitions into account.

The third group of approaches uses abstract behaviour models of the application to generate meaningful test cases. These test models are created manually, generated from source code or derived from other models through model transformation [7]. Walkinshow et al. [8] trace the execution of software to infer a test model. Different modelling languages including state charts, Petri nets, message sequence charts or Finite State Machine (FSM) can be used [9]. While executing a test case an execution engine iterates through the elements of the model, e.g., a transition in a FSM, to trigger the SUT and validate its output. Since the efficiency of the test case highly depends on the test data, a lot of research has be done in the field of test data generation. The challenge is to find boundaries of the valid input space. Tracey et al. [10] exploit search techniques to automate the generation of test data. Evolutionary algorithms, namely Genetic Algorithm (GA) are used to derive test data from an initial data pool in [11] to have a fully automated test data creation. Here, a new generation has close relations to the generation before, thus focusing on relevant test data. Fischer et al. [12] propose the use of a GA to enhance the quality of automatically generated test data. IoT-based services are often based on energy restricted (e.g., battery driven) sensors and actuators that have a limited number of usage cycles. This IoT limitation hinders GA usage in testing due to the high amounts of test cases, which are used to optimise the test data. To overcome this limitation the proposed approach realises the optimisation of test data and test cases by using service descriptions before the service is tested.

In recent years several works investigated model based testing approaches for services. Ramollari et al. [3] create functional conformance tests by utilising IOPE sets without detailed interface descriptions. A semantic parameter conformance validation can be found in [13] missing the abstraction of detailed interface parameters and following a stateless approach. The commercial available solution [14] enables functional test creation based on web service interface descriptions. The approach of Schanes et al. [15] concentrates on Extensible Markup Language (XML) as generic data format for test execution. They both do not consider stateful service behaviour of reactive systems by modelling conjunctions between various methods of the service.

III. Concept and Architecture

The presented concepts are part of the IoT.est [16] project, which aims at developing an IoT service creation environment whilst bridging the gap between various business services and the heterogeneity of networked sensors, actuators and objects. The approach employs semantic service descriptions to compose IoT services and derive corresponding functional conformance tests, semi-automatically. After the manual annotation of a service the service model is generated automatically. It can be altered manually before the automated generation of test cases begins. A consistent service concept is specified to enable this process.

A. IoT.est Service Concept

IoT.est utilises RESTful interfaces to encapsulate IoT services for enhanced re-usability. It defines two types of services to ensure direct consumption and composition of IoT services without dealing with heterogeneous interfaces:

The Atomic Service (AS) is a RESTful web service, accessing $0 - n$ IoT resources via their own individual interfaces and radio technologies. It enables access to these resources via standardised `Get, Post, Put, Delete` request methods, whose invocation is defined in a Web Application Description Language (WADL) document [17]. Input parameters as well as service responses are extensively semantically described in the Knowledge Management (KM). The implemented AS can be deployed to a Runtime (RT) for web services and is registered in the KM.

The Composite Service (CS) enables a business process-based composition of various AS and CS. It also provides a RESTful interface for service invocation and does not directly connect to IoT resources using their proprietary interfaces. It only uses AS and CS interfaces to acquire sensor information and to control actuators. The interfaces are also described in WADL and a semantic description is used to enable re-usability for composition and testing.

B. Architecture

The IoT.est project architecture specifies a Test Design Engine (TDE), which enables the generation of test data and derivation of test cases and flows for IoT services (see Figure 1). The derivation is driven by processing service descriptions and utilising domain knowledge. IoT.est uses a KM to store descriptions of IoT services. These services can be deployed and composed via a Service Composition Environment (SCE) in distributed RT environments of the framework. To support testing by the Test Execution Engine (TEE) prior to runtime deployment we employ a Sandbox Runtime (SRT) instance of the RT. The SRT supports emulation of IoT resources [18] to enable IoT service testing without communicating with resource constrained IoT sensors or altering IoT actuators during test execution.

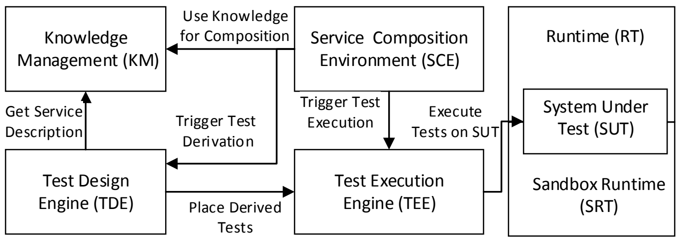


Figure 1. Simplified IoT.est Architecture.

To obtain a comprehensible test generation, the TDE utilises an explicit information representation approach, which can also be used to evaluate and alter the model and tests, which are automatically derived. During the first step the service model, which is generated from the semantic description, is represented as an Eclipse Modeling Framework (EMF)-model. It is editable with the Graphical Modeling Framework (GMF). During the second step, test cases are created in the ETSI standardised Testing and Test Control Notation Version 3 (TTCN-3) to obtain a readable and reusable representation.

IV. Service Interface Description

In this section, the different types of service descriptions are described. These descriptions are used in Section VI to build the EMF state machine model. The client–server communication of RESTful services is constrained by no client context being stored on the server between requests [19], although services can follow a stateful behaviour. Since the interfaces are implemented stateless there is a missing support of behavioural descriptions in established description notations like WADL. To enhance testability the proposed approach extensively describes service interfaces and also the service behaviour to get information for valid and invalid interface calls with test parameters depending on parameter values and current service states. The information is used to enable an ECP-based model generation.

A. Precise Parameter Descriptions

The service model creation utilises service descriptions to find valid and invalid equivalence classes, which are used to model state-based transitions. The equivalence classes are processed by a boundary value analysis and random value generators to derive the test cases. Conventional service descriptions, based on WADL, describe resource parameters as implementation-specific technical parameters using well known data types like `string` and `double` (shown in Figure 2). This leads to a very simple equivalence class model, which accepts the whole data type as valid input although the application specific usage of the parameter can be restricted to a small range.

```
<resource path="/zoom/{id}/{value}">
  <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="
      id"
   style="template" type="xs:string" />
  <param name="value" style="template" type="xs:double" />
  <method id="setZoom" name="POST" />
</resource>
```

Figure 2. Basic WADL Parameter Descriptions.

The following paragraphs show examples of information, which is used to precisely define service parameters.

1) Simple Value Range Limitation: A fundamental approach of the enhanced service descriptions is to define the precise value ranges of parameters to gain an abstracted model of method parameters. This model is not only based on a technical data type that is used to transfer the information. It also specifies the defined value ranges processed by the service logic (e.g., a valve position between -25.0 and 15.5). A simple limitation of this parameter value in an XML-Schema is shown in Figure 3. Numeric data types can be restricted by value ranges and an enumeration of allowed values. Character data types can be restricted by the number of allowed characters, the length, an enumeration or a regular expression which could e.g., define an email-pattern. The mapping between a parameter of a method or resource in the WADL file is performed by namespaces, which do not require any extension of the existing WADL definition.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="valve">
  <xs:simpleType>
    <xs:restriction base="xs:double">
      <xs:minInclusive value="-25.0"/>
      <xs:maxInclusive value="15.5"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Figure 3. Simple XML parameter restrictions.

The given example describes one valid (vP) and two invalid equivalence classes (iP). Since division by zero and switching between negative and positive values are typical code weaknesses, we divide the valid class into two using $-0, +0$ for boundary value analysis (see Figure 4). This methodology results in 4 disjoint classes: iP_1, vP_1, vP_2, iP_2

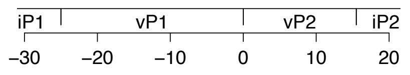


Figure 4. Equivalence Class Partitioning Example.

The definition of valid partitions is not only limited to a single value range it can describe various valid and invalid partitions for one parameter (see XML-Schema restrictions [20]). It also supports complex definitions of strings not only by enumerations but also with regular expressions. This way, e.g., an email address can be described as parameter input. The definition of regular expressions is then reused for the test data generation.

2) Semantic Parameter Description: The test data generation uses semantic annotations that can be linked to upper level ontologies like SUMO [21] for reusable test case derivation. Reusable parameter limitations can, e.g., restrict the range of a Celsius temperature and the possible temperature units or define e.g., sets of countries.

Semantic Parameter Interdependency: The description of service parameters has to take into account that they have interdependent connections to each other. Ontology documents take this into account by describing individuals using classes, relations and attributes. The value range of a parameter *Cityname* for example depends on the *Countryname* parameter since for example the city *Bologna* exists in *Italy* but not in *Germany*. The description of linking interdependent parameters on the predicate *geographicSubregion* is shown in Figure 5. The *owlType* definition is declared for each semantic parameter and linked to a class definition within the ontology by the *requestLink* tag (Figure 5:3,6). The *restriction* tag describes the predicate on which the interdependency is defined (Figure 5:7). Figure 6 shows the generated SPARQL Protocol and RDF Query Language (SPARQL) code that is used to find the matching entities in the ontology.

```
1 <owlTypeDefinition>
2  <owlType name="Countryname" type="base">
3   <requestLink uri="http://www.onto.org/SUMO.owl#Nation"/>
4  </owlType>
5  <owlType name="Cityname" type="restricted">
6   <requestLink uri="http://www.onto.org/SUMO.owl#City"/>
7   <restriction uri="http://www.onto.org/SUMO.owl#
        geographicSubregion" value="{Countryname}"/>
8  </owlType>
9 </owlTypeDefinition>
```

Figure 5. Semantic Parameter Interdependency.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix sumo: <http://www.ontologyportal.org/SUMO.owl#>
select ?city where {
  ?city rdf:type sumo:City .
  ?city sumo:geographicSubregion sumo:Germany .
}
```

Figure 6. SPARQL-Query.

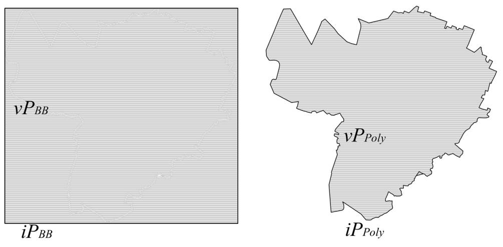
B. Geospatial Testdata Derivation

Since IoT-based sevices often cover specific areas, a geospatial description of services is very useful to determine functional conformance. A common description approach for geospatial areas is to describe a bounding box (rectangle) defining the min. and max. latitude and longitude values that cover an area. This often leads to a very imprecise area description. A better way is to specify the precise geospatial coverage by defining a polygon(concatenation of a list of coordinates, surrounding an area), which defines the covered area. Since for a complex polygon like a city boundary it's a very long description it is not feasible that everybody annotates a precise polygon for every supported area. Therefore, we use an external knowledge base (OpenStreetMap (OSM) [22]) that can access those polygons just by annotating it with the city/country name. The following shows an example of the three annotation methods. You can see the resulting areas in Figure 7.

- Bounding Box (BB):
  longitude *min:11.2295654 max:11.4336305*,
  latitude *min:44.4211136 max:44.5566394*
- Polygon (544 coordinates): *(11.366030, 44.449526 11.363828, 44.450242 11.362467, 44.450953 11.362356, 44.451083 11.360538, 44.44932 . . .)*
- Country name and city name lookup in spatial data infrastructure (OSM): *Italy, Bologna*

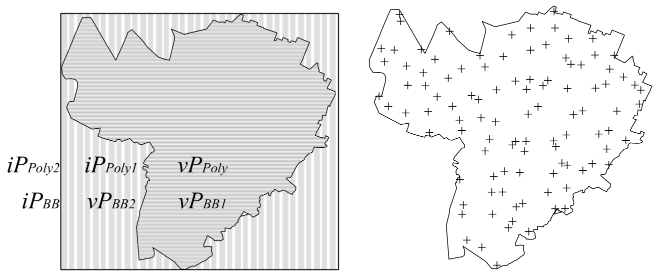
Figure 7 shows the created equivalence partitions of the bounding box (Figure 7(a)) and the polygon (Figure 7(b)). The precision improvement of the polygon is shown in Figure 8(a) since after comparison it shows that the bounding box describes a false positive valid equivalence class (see Equation 1):

$$vP_{BB2} \backsim iP_{Poly1} \tag{1}$$



(a) Equivalence Classes by BB (b) Equivalence Classes by Polygon

Figure 7. Equivalence Partitioning of the Bologna Area.



(a) Comparing the Equivalence Classes (b) Random Generated Test Data

Figure 8. Utilising OSM for ECP.

Figure 8(b) shows the test data generation creating 102 coordinates for randomly testing the service in the covered area. Using boundary value analysis on the buffered polygon it is also possible to test the service behaviour at the defined border with valid and invalid values.

C. Service Behaviour Description

By featuring clearly defined interface and parameter descriptions, documents such as WADL enable easy technical integration of RESTful services. The lack of standardised stateful service descriptions is the main challenge for the process of stateful testing. Therefore, a simple and easy to use description format has been developed which can be used to define typical use cases for the IoT service. In addition to the existing WADL document, this description format facilitates the definition of a sequence of resource and method executions including loops and concurrent calls. The XML-based sequence description document refers to method calls in the WADL document to gain compatibility. Figure 9 illustrates the main structure of a sequence description document.

```
1 <sequencespecification xmlns:ws1="application.wadl">
2  <vars><var name="cameraPan" type="double"/></vars>
3  <paramsets><paramset id="cameraPan">
4   <param name="id">10.11.127.6</param>
5   <param name="value"></param>
6  </paramset>...</paramsets>
7  <results><result name="testPosition" mediatype="application
        /xml">...</result>
8  </results>
9  <sequence mode="multiple" subSequenceType="All">
10  <subsequence mode="single" subSequenceType="
        MutualExclusive">
11   <subsequence mode="single">
12    <wsuri path="ws1:/Camera/pan/{id}/{value}" paramset="
         cameraPan"/>
13    <wsmethod name="ws1:setPan" returnCode="2xx"/>
14    <setvar var="cameraPan">{value}</setvar>
15  </subsequence>
16  <subsequence mode="single">...</subsequence>
17 </sequence>
18 </sequencespecification>
```

Figure 9. Service Sequence Description.

The sequence and subsequence elements are transformed into a state machine to enable the model based testing process. Each sequence and subsequence with a *wsmethod* and *wsuri* definition represents a single state and at least one transition to this state in the constructed state machine. The number of transitions depends on the number of values for the parameters and the combination of the same. The elements are structured into groups, which will be affecting the structure of the state machine directly (e.g., multiple paths, creation of sub state machines). Each sequence has a definition of a called WADL-resource (*wsuri*, Figure 9:16) and a method (*wsmethod*, Figure 9:18), which is provided by the IoT service. Furthermore, control commands are also a part of a sequence. In case of the control command *setvar* an actual value of a parameter from a method or a resource is saved into an internal variable. This procedure allows the implementation of stateful knowledge since the internal variable can be used over different transitions at any time and can also be part of a validation process. The content of a response message of an IoT service is mapped on a result definition to validate the content against previous inputs. With the sequence description, the values for each parameter (e.g., resource and method) can be predefined for this use case by the user. Missing values for each individual simple or complex (e.g., structures like XML) parameter in the sequence description are generated by the test data generation if a user provides only a portion of parameter values for the use case.

V. Evaluation of Example Service

To demonstrate the algorithms and the process of the test framework this work employs a camera control example service. The service is used to control multiple CCTV Cameras at different locations that are adjustable in their pan and tilt via a RESTful interface. The following sections describe the transformation and test derivation aligned to this service. The sequence begins with an initialisation process of the camera (illustrated in Figure 10).

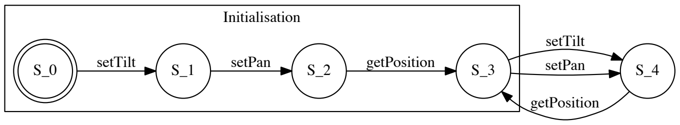


Figure 10. Camera Example Service.

Between S_3 and S_4, the values can be set with the `setTilt` and `setPan` method and evaluated with the `getPosition` method at the transition back to S_3.

VI. Test Case Derivation & Execution

The retrieval of the outlined service behaviour description of Section IV-C is the starting point for the test case derivation and execution, which is explained in this section. Whereby the main aim is to enable a fully automated test derivation process it also allows manual enhancements based on expert knowledge. The approach is divided into two translation steps: *i)* derive an EMF service model that represents an abstract behaviour of the SUT from testing perspective (e.g., detectable behaviour) and *ii)* derive executable test cases from this model based on TTCN-3. While the two steps are fully automated, the test developer can adapt the derived EMF service model with an Eclipse GMF editor and the created TTCN-3 test cases. For the model transformation, classical state machine concepts of states and transitions are re-used. In addition, the inclusion of concepts of TTCN-3 (e.g., Ports, Components, MessageTemplates) enables an easy model transformation. The basic model objects are shown in Figure 11 and can be described as follows:

States represent different logical conditions of the SUT and limit the number of correct functionality.

Events characterise the starting of an activity, which might result in actions or a state change. Events can be either from the type timer or input message.

Actions describe the reaction of the system to an event. An Action can be either a response message (output) or can result in a request sent to an IoT resource.

Transitions describe how the SUT reacts (action) to a certain event and a specific state. A Transition connects different states within the model.

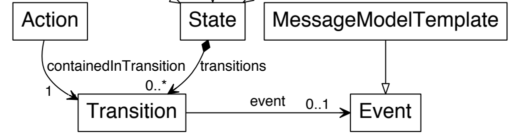


Figure 11. Simplified EMF Service Model.

The stages of the test derivation process and its information flow are shown in Figure 12. The process is initiated via a web interface during stage A. At this time, the required service behaviour descriptions are retrieved from the Knowledge Management. The service descriptions are analysed and transferred into the EMF model in stage B. At stage C this model is translated into executable test cases (TTCN-3). The final stage D executes these test cases and evaluates the results. The following paragraphs explain this process in more detail.

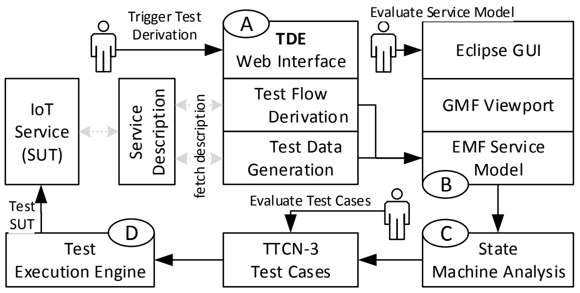


Figure 12. Information Flow.

A. Start of the Test Derivation Process

The test derivation process is triggered via a web interface, which accepts an ID identifying the service that was registered and has to be tested. The TDE fetches the needed service description documents from the KM and evaluates links of the semantic annotations to build the complete data model.

B. Building the EMF Service Model

The *Test Data Generation* analyses referenced data types and their interdependencies, which are described in the service descriptions. For the derivation of test data for each parameter ECP is used since it has been proven to provide high effectiveness in finding defects [23]. This technique divides the possible input data for each parameter in at least two disjunctive partitions (e.g., valid and invalid values). The partitions are created by parsing the parameter restrictions (see Section IV). The test data generation is designed to cover each partition with at least one test case. Due to the behaviour change of an IoT service between the boundary of two disjunctive partitions, the test data generation uses a boundary-value analysis to fully cover the boundaries of each partition. In this approach, valid parameter ranges are annotated with the use of XML Schema and with a semantic description. It generates code snippets for parameterised method invocations for the RESTful service and stores them in the EMF service model. Those snippets are based on generated random parameters for the used data types or enable code libraries based on lazy testing [24] to generate test data during runtime.

The *Test Sequence Analysis* is used to build the EMF service model based on the IoT Service WADL description and a sequence description. The model is implemented as a Extended Finite State Machine (EFSM), which has at least a unique InitialState and an EndState, as well as one NormalState definition. While the InitialState and the EndState(s) are generated automatically, the NormalStates will be created through the process shown in Table I. If a sequence has subsequence definitions multiple Normal states are created, followed by the creation of state transitions, which connect two

TABLE I. DOCUMENT TRANSFORMATION TO MODEL OBJECT NORMALSTATE.

Input:	Action:
<sequence mode=”single”> ... </sequence>	Create NormalState for each sequence part.
<!−− WADL document −−> <resources base=”http://10.1.1.42:80/CameraService/iot/”> <resource path=”/Camera”> <resource path=”/pan/{id}/{value}”> <param name=”id” style=”template” type=”xs:string”/> <param name=”value” style=”template” type=”xmlschema:pan”/> <method id=”setPan” name=”POST”/> </resource> </resource> </resources> <!−− Sequence description −−> <sequence mode=”single”> <wsuri path=”ws1:/Camera/pan/{id}/{value}” paramset=”cameraPan”/> <wsmethod name=”ws1:setPan” returnCode=”2xx”/> </sequence>	Create Message-ModelTemplate (MMT) event for a request to a IoT service.

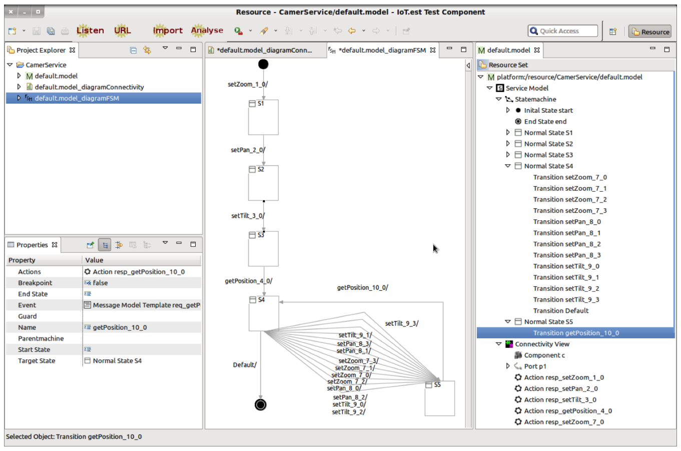
<paramset id=”cameraPan”>
  <param name=”id”>10.11.127.6</param>
  <param name=”value”>12</param>
</paramset>
	Generate test data or use user defined values from sequence for each parameter.

<vars> <var name=”cameraPan” type=”double” schema=”response:PositionResponse#pan”/> </vars> <sequence mode=”single”> <wsuri path=”ws1:/Camera/pan/{id}/{value}” paramset=”cameraPan”/> ...<setvar var=”cameraPan”>{value}</setvar> </sequence>	Create variables which will represent the actual used value for a parameter. Add to MMT event.
<method id=”getSensingData” name=”GET”> <response> <representation mediaType=”application/xml”/> </response> </method>	Create a MMT action for the response of a IoT service.
<wsmethod name=”ws1:setPan” returnCode=”2xx”/>	Create and add range of expected HTTP status code to MMT action.

<vars>
 <var name=”cameraPan” type=”double”
  schema=”response:PositionResponse#pan”/>
  ...</vars>
<results>
  <result name=”testPosition” mediatype=”application/xml”
   type=”xml”>{cameraTilt,cameraPan}</result>
<results>
<sequence mode=”single”>
  <wsmethod name=”ws1:getPosition”
   return=”responseVar” result=”testPosition”/>
</sequence>
	Add handling of possible return values to the MMT action (e.g., xml structure as response). Define variables to save the return values.

different states. These transitions represent an interaction with the IoT service or a timer Event. Therefore, the transaction needs a definition for both parts of the communication (e.g., request and response). The request is represented by a MMT event, which is the following step for the transformation. The resource and method information are part of the sequence definition and are extracted from the linked WADL document. If the sequence does not specify user defined parameter values, the test data generation will produce test values for each parameter.

A sequence definition is enabled to host control commands like the *setvar* tag listed in Table I/row 4. It is used to save the current value of a parameter for future operations and



Property	Value
Actions	Action resp_getPosition_10_0
Breakpoint	false
End State	
Event	Message Model Template req_getP
Guard	
Name	getPosition_10_0
Parentmachine	
Start State	
Target State	Normal State S4

Selected Object: Transition getPosition_10_0

Figure 13. EMF Model Evaluation in Eclipse.

thereby allows the integration of stateful knowledge into the state machine representing a data model of the IoT service. The response template of the IoT service is modelled as a MMT action storing the method and response representation, which are defined in the WADL document that describes the service (I/row 5). Furthermore, the sequence description defines the expected HTTP status code for the response. The result attribute for a *wsmethod* tag in a sequence description (I/row 7) indicates another control command for the algorithm of the test case derivation and execution. This command produces a mechanism to compare the response of the IoT service against previous used parameter values, which can be used as a decision if the actual transition is valid or invalid. The MMT action as well as the event is be linked to a single transition in the resulting state machine model.

After creation the EMF model can be evaluated and altered in an Eclipse GMF - Model Editor (see Figure 13).

C. Creating Test Cases From the Service Model

The EMF model is used to analyse the resulting state machines of the previous stage. A configuration of the test case generation allows transition coverage or transition and state-coverage based on the W method [25]. The transition coverage is computed by identifying the *InitalState* and building a test tree based on a breadth-first visit of all transitions. Each transition in each state is inspected and if the transitions directs to a unvisited state a new branch path is created. Afterwards, the new branch end states are visited and their transitions are inspected. Each new inspected transition results in a new test path, which represents the test cases if only transition coverage is selected. For transition and state coverage it is further needed to identify a characterisation set (also called W set), which is a set of input sequences that can be utilised to distinguish every pair of existing states in the model. The resulting test cases are created by concatenating every sequence from the transition coverage set with every sequence in the characterisation set and apply them after the SUT is initiated.

The Model transformation from EMF to standardised TTCN-3 ensures explicit representation and reproducibility of test cases. As output of stage C test cases are derived from the EMF Service Model. A test case is a directed graph consisting of states and transitions and represents one possible path from the InitialState to another defined NormalState or

EndState (e.g., $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_3$, see Figure 10). During the model transformation each element is inspected and the required TTCN-3 elements are created. The actual writing of the TTCN-3 code is realised with a template engine. This enables the separation of syntactical details of the TTCN-3 language from the analysing logic thus reducing the complexity and enhancing the manageability. The followed approach uses the Java-based template engine Velocity [26]. In the following the transformation step is outlined with some detail. Table II depicts the first step while going through the model elements in the current test case. The model object *InitialState* is used to create the general test case structure and assures that the test case stops after a defined time by adding a timer. Afterwards, the TTCN-3 element *function* is created and added to the test case. TTCN-3 functions are utilised to separate different steps of the test execution. These reusable functions are used to represent the different states of the SUT.

TABLE II. TTCN-3 TRANSLATION OF MODEL OBJECT INITALSTATE.

Action:	TTCN-3 Output:
Add timer to enable timout	testcaseMaxExecutionTimer.start;
Create function	function start_1_0() runs on c { ...}
Add function to Test Case	testcase tc_1() runs on c system sys { start_1_0(); ... }

The next element Transition consists of an Event that can describe that an input is received by the SUT and an Action that describes the output reaction of the SUT to this input message. Table III sketches the transformation from the model object event to a send operation and the storage of the sent values for later usage. Since the EMF service model is created from the service point of view the translator inverts certain expressions for the purpose of testing. In this case the event of a transition becomes a send call.

TABLE III. TTCN-3 TRANSLATION OF MODEL OBJECT EVENT.

Action:	TTCN-3 Output:
Create send call and local variable	template HttpRequest req_setPan_1_0 := { postRequest := { url := "http://10.1.1.42:80/CameraService/iot/Camera/pan /10.11.127.6/19.27", ... } } v_PositionResponse_pan := 19.27; f_request(p1, req_setPan_1_0); v_req_setPan_1_0 := req_setPan_1_0;

Subsequently, the action part of the transition is utilised to derive TTCN-3 code. Initially a new function for the next state is created. Afterwards the defined response of the SUT is translated into TTNC-3. Then, the TTCN-3 element *alt* is used to form the possibilities of the SUT behaviour. At first, the failure case for delayed or unexpected service responses is modelled. After that the followed approach assumes deterministic service behaviour with only one possible valid reaction. This expected behaviour is included in the *alt* element of TTCN-3 including the jump to the next TTCN-3 function (state) created before. Table IV shows the discussed transformation process of the model object action.

While the link to the next function has been created during the action transformation, in the last step the function itself is created at the time the next element (*NormalState*) of the test case is inspected. Table V reveals the resulting TTCN-3 output.

At the final stage of the test case the model, object *EndState* is reached. This completes the TTCN-3 code creation by

TABLE IV. TTCN-3 TRANSLATION OF MODEL OBJECT ACTION.

Action:	TTCN-3 Output:
Create Target Call	S1_1_2();
Create expected response message	var template GETResponse resp_setPan_1_0 := { statusCode := (200 .. 299), content := {rawContent := omit, plainTextContent :=?}, headers := ? }
Form alt for Message	alt { [] testcaseMaxExecutionTimer.timeout { tcMaxExecutionTimeout_1();} [] any port.receive { unexcepctedStateReached_1(); } }
Create reply element in alt	alt { [ischosen(req_setPan_1_0.postRequest)] p1.getreply(POSTreq: {req_setPan_1_0.postRequest} value resp_setPan_1_0) −> value v_resp_setPan_1_0 { S1_1_2();} ...}

TABLE V. TRANSLATION OF MODEL OBJECT NORMALSTATE.

Action:	TTCN-3 Output:
Create function	function S1_1_2() runs on c {...}

setting the verdict to pass. If all functions, corresponding requests and response messages have been transmitted during the test case execution this final statement indicates that the SUT has the expected behaviour for this test case. Table VI shows the resulting TTCN-3 code.

TABLE VI. TTCN-3 TRANSLATION OF MODEL OBJECT ENDSTATE.

Action:	TTCN-3 Output:
Set verdict	setverdict(pass, "End−state reached");

D. Executing the Test Cases

After compilation of the TTCN-3 test cases the whole test flow can be executed by a web service interface or manually using the TTworkbench [27]. It enables a visualised logging of test execution in a log report, which can be used to evaluate the detailed test results (see Figure 14).

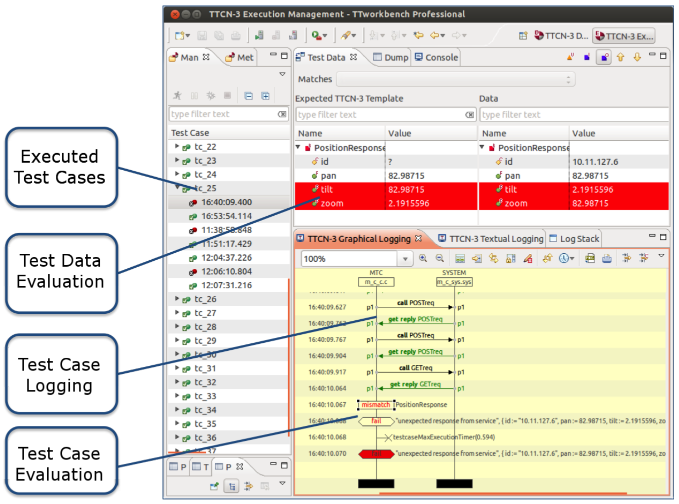


Figure 14. Execution of the Test Cases.

The TTworbench provides a comprehensible graphical view to easily identify the cause of an occurred error (e.g., protocol, encoding or data).

VII. Conclusion and Future Work

The complexity to describe IoT services for testing purposes in conjunction with missing domain specific knowledge for data types has prevented the utilisation of automated model-based testing for IoT services. The outlined framework tries to lower the gap by employing a sequence based modelling description which can be easily created, whereby the automated state machine analysis allows a transition and parameter combination coverage. Utilising semantically definitions in combination with ECP provides distinct test data pools enabling a more efficient and domain specific test case generation. The testing framework follows a two-step approach where the service description includes common utilisation information within a sequence description. The combination of standardised WADL interface description, semantic parameter descriptions and a sequence description empowers the transformation into a service model. Afterwards test cases can be derived and executed based on TTCN-3. This approach enables adjustments by developers at an early stage due to simple sequence descriptions and the standardised test notation TTCN-3. The key principles of the test framework are explained based on an example IoT service. The example is directly taken from our prototypical implementation and proves the applicability of our approach for IoT services. Although there is a high complexity in the initial implementation of the framework, the automated derivation allows the tester to take a systematic model driven approach to test IoT services though keeping possibilities to evaluate and modify the created test cases in a standardised test notation. The implemented sequence definition fills the gap between stateless interface descriptions and model-based testing and can be used for a more simplified and controllable test automation.

As a common approach IoT service compositions are utilising high level business modelling languages like Business Process Model and Notation (BPMN) [28]. Therefore, future work will include the integration of such languages and annotate them semantically to enable automated derivation of a service model. Besides functional behaviour, the influence of networking and service quality characteristics needs to be addressed for large scale IoT service testing.

Acknowledgment

The research leading to these results has received funding from the European Union FP7 for the IoT.est project under grant agreement n° 257521.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," Computer networks, vol. 54, no. 15, May 2010, pp. 2787–2805.

[2] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Software Testing, Verification and Reliability, vol. 22, no. 5, 2012, pp. 297–312.

[3] E. Ramollari, D. Kourtesis, D. Dranidis, and A. Simons, "Leveraging semantic web service descriptions for validation by automated functional testing," The Semantic Web: Research and Applications, Jun. 2009, pp. 593–607.

[4] D. Binkley, "Source code analysis: A road map," in 2007 Future of Software Engineering, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 104–119.

[5] A. Takanen, Fuzzing for software security testing and quality assurance, ser. Information security and privacy series. Artech House, 2008.

[6] W.-l. Huang and J. Peleska, "Exhaustive model-based equivalence class testing," in Testing Software and Systems, ser. Lecture Notes in Computer Science, H. Yenign, C. Yilmaz, and A. Ulrich, Eds. Springer Berlin Heidelberg, 2013, vol. 8254, pp. 49–64.

[7] E. G. Aydal and J. Woodcock, "Automation of model-based testing through model transformations," in Testing Conference - Practice and Research Techniques, 2009. TAIC PART '09. IEEE, Sep. 2009, pp. 63–71.

[8] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," in 2013 20th Working Conference on Reverse Engineering (WCRE), Oct. 2013, pp. 301–310.

[9] A. Pretschner, "Model-based testing," in 27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings, May 2005, pp. 722–723.

[10] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in Automated Software Engineering. 13th IEEE International Conference on, Oct. 1998, pp. 285–288.

[11] M. Deng, R. Chen, and Z. Du, "Automatic test data generation model by combining dataflow analysis with genetic algorithm," in Pervasive Computing (JCPC), 2009 Joint Conferences on, Dec. 2009, pp. 429–434.

[12] M. Fischer and R. Tonjes, "Generating test data for black-box testing using genetic algorithms," in 2012 IEEE 17th Conference on Emerging Technologies Factory Automation (ETFA), Sep. 2012, pp. 1–6.

[13] K. Belhajjame, S. Embury, and N. Paton, "Verification of semantic web service annotations using ontology-based partitioning," IEEE Transactions on Services Computing, vol. 99, no. PrePrints, 2013, p. 1.

[14] C. Kankanamge, Web Services Testing with SoapUI. Packt Publishing Ltd, 2012.

[15] C. Schanes, F. Fankhauser, S. Taber, and T. Grechenig, "Generic data format approach for generation of security test data," in VALID 2011, The Third International Conference on Advances in System Testing and Validation Lifecycle, Oct. 2011, pp. 103–108.

[16] 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 Future Network and Mobile Summit, Berlin, 2012, pp. 1–8.

[17] M. J. Hadley, "Web application description language (wadl)," Sun Microsystems, Inc., Mountain View, CA, USA, Tech. Rep., 2006.

[18] E. Reetz, D. Kuemper, K. Moessner, and R. Tönjes, "How to test iot-based services before deploying them into real world," in 19th European Wireless Conference (EW 2013), Guildford, United Kingdom, Apr. 2013, pp. 1–6.

[19] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.

[20] P. Biron and M. Ashok, "Xml schema part 2: Datatypes," W3C Recommendation, vol. 2, 2001.

[21] I. Niles and A. Pease, "Towards a standard upper ontology," in Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001, ser. FOIS '01. New York, NY, USA: ACM, Oct. 2001, pp. 2–9.

[22] A. Ballatore, M. Bertolotto, and D. C. Wilson, "Geographic knowledge extraction and semantic similarity in openstreetmap," Knowledge and information systems, vol. 37, no. 1, Oct. 2013, pp. 61–81.

[23] N. Juristo, S. Vegas, M. Solari, S. Abrahao, and I. Ramos, "Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects," in Software Testing, Verification and Validation (ICST), 2012 IEEE, Apr. 2012, pp. 330–339.

[24] M. Lin, Y. Chen, K. Yu, and G. Wu, "Lazy symbolic execution for test data generation," Software, IET, vol. 5, no. 2, Apr. 2011, pp. 132–141.

[25] A. Gargantini, "4 conformance testing," in Model-Based Testing of Reactive Systems. Springer, 2005, pp. 87–111.

[26] Apache Software Foundation, "The apache velocity project," Website, available online at http://velocity.apache.org/ retrieved: 2014-08-30.

[27] Testing Technologies, "TTworkbench," Website, available online at http://www.testingtech.com retrieved: 2014-08-30.

[28] S. Meyer, A. Ruppen, and C. Magerkurth, "Internet of things-aware process modeling: Integrating iot devices as business process resources," in Advanced Information Systems Engineering, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7908, pp. 84–98.

A Novel Approach for Environment Model-Based Functional Testing of Reactive Systems

Annamária Szenkovits and Hunor Jakab
Faculty of Mathematics and Computer Science
Babes-Bolyai University
Cluj Napoca, Romania
Email: {szenkovitsa, jakabh}@cs.ubbcluj.ro

***Abstract*—Automating the test process of safety-critical reactive systems is an important problem in the software testing domain. One of the major difficulties in achieving this is that test sequences cannot be generated without feedback from the environment due to the reactive nature of the system. A common solution is to model the environment and manually fine-tune the model to produce test cases that target specific important usage patterns. This paper presents a novel approach to environment-based functional testing that automatically performs the tuning of the environment model such that the generated test cases cover important regions of the input space. Our method is based on evolutionary techniques with the goal of optimizing the weights associated with choice nodes and variable bounds in an environment model written in the Lutin language. An experimental test-bed is proposed based on SCADE models of the Transmission Beacon Locomotive 1 (TBL1) system to validate our approach in a realistic environment.**

Keywords–Reactive systems; Environment model-based testing; Evolutionary testing.

I. Introduction

Reactive systems are in continuous interaction with their environment. They control the environment, and must also react to the stimuli of the environment within a given time bound. Therefore, in order to be able to automatically generate test sequences, we must also simulate the environment and the interaction between the environment and the System Under Test (SUT). In order to detect possible faults in the SUT, we must drive the environment in such configurations that might violate some safety properties of the SUT. This is however not a simple problem, since it requires knowledge from domain experts.

A common way to express the properties of a reactive system is to describe the model of the system in Lustre, a language optimized for reactive systems [1][2][3]. Lustre is also the kernel of the Safety Critical Application Development Environment (SCADE) [4], a widely used industrial toolset. The models of the TBL1 system, proposed for the experimental validation of our methods, were also implemented using SCADE. As for environment models, a convenient way to model the environment of Lustre and SCADE models is to use the Lutin language [5], an automatic test generator for reactive programs that focuses on functional testing.

This paper presents a work in progress which is based on a method that automatically fine-tunes the environment model in order to generate test scenarios that might detect faults in the SUT. We propose a solution to the problem of fine-tuning the environment model based on evolutionary techniques [6]. More precisely, we are going to exploit some of the features of the Lutin language in order to optimize the generation of test scenarios.

The paper is structured as follows. Section I-A reviews some of the work relevant for this topic, Section II briefly describes the behavior of reactive systems and the difficulties that arise in case of test input generation for reactive systems. Parts II-A and II-B present some of the fundamental aspects of the languages Lustre and Lutin, respectively, focusing on how different properties of these languages will be exploited by our method. Part II-C summarizes how evolutionary algorithms are planed to be used for environment optimization. Finally, Section III describes the TBL1 system.

A. Related Work

Our work is related to environment model-based testing of reactive systems, as well as to evolutionary testing, two important research domains that have been explored in a number of references. We mention a few of the related articles which emphasize the practical applicability of evolutionary methods to real-life problems. The work in [7] discusses the scalability, applicability, and acceptability of evolutionary functional testing in industry. The problem is investigated through two case studies, drawn from serial production development environments. The methods presented by Corno et al. [8] and Iwashita et al. [9] use an evolutionary algorithm to automatically generate a test program for pipelined processors by maximizing a given verification metric. Genetic Evolutionary Algorithms (GEA) are also used for test generation by Cheng and Lim [10]. The problem of parameter selection is discussed and a Markov chain based method is used to model the test generation process and to parametrize the process characteristics. The method is used here in particular for generating test cases to verify hardware design for semiconductor industry. However, unlike our approach, the methods discussed in the above mentioned papers are not optimized for reactive systems.

There are several tools available for performing model-based testing on reactive systems. Bousquet et al. [11][12] present a specification-based language called Lutess, while Marre et al. [13][14] describe Gatel, a test generation tool for Lustre programs. Our approach is based on similar principles, with the added benefit of being able to optimize the distribution of the generated test cases. This can be crucial in complex systems where exhaustive testing is infeasible and specific usage scenarios need to be targeted.

II. Functional testing of reactive systems

Reactive systems have cyclic behavior, meaning that at each cycle they read the inputs coming from their environment,

```
node never (A: bool) returns (never_A: bool);
let
never_A = not(A) -> not(A) and pre(never_A);
tel
```

Figure 1. Example of Lustre code.

compute the outputs and update the internal state of the system. Considering this, instead of generating a single test input, the tester has to provide test sequences, i.e., sequences of input vectors.

Another issue that arises during test input generation is that input sequences cannot be generated offline. Because a reactive system is in continuous interaction with its environment, the input vector at a given reaction may depend on the previous outputs. Thus, input sequences can only be produced on-line, and their elaboration must be intertwined with the execution of the SUT.

Due to the above seen properties, programming reactive systems is not easy in conventional languages. Lustre [1][2][3], on the other hand, is a synchronous languages, which means that it is optimized for reactive systems. Therefore, it is more suitable to implement the cyclic behavior of such systems. Lustre is also the backbone of SCADE, a tool widely used in the railway, automotive and aviation industry. The models proposed for the validation of our methods were also implemented in SCADE. In this article we are going to use Lustre for the description of the SUT. This section provides a brief overview of the language's structure which are key to understanding the rest of the paper.

A. *Describing the SUT properties*

Lustre is a synchronous language based on the data flow model and designed for the description and verification of reactive systems [1][2]. It can be used for both writing programs and expressing program properties. It is structured on so-called *nodes*, where a node represents a program or a subprogram and it operates on *streams*: a finite or infinite sequence of values of a given type. A program has a cyclic behavior, so that at the nth execution cycle of the program, all the involved streams take their nth value. A node defines one or several output parameters as functions of one or several input parameters. All these parameters are streams.

Figure 1 shows an example [3] of a Lustre node.

The node defined in this example takes as input the Boolean stream $A = (A_1, A_2, ..., A_n, ...)$ and defines as output another Boolean stream $never\_A = (never\_A_1, never\_A_2, ..., never\_A_n, ...)$. The output is true if and only if the input has never been true since the beginning of the program execution.

Assertions can be also included into the body of a Lustre program. They are boolean expressions that should be always true. Safety properties, the properties of a program's environment can be easily specified by using the assertion mechanism. Assertions will be exploited in our method for driving the SUT environment as close as possible to configurations that might reveal failures in the SUT.

B. *Modeling the environment*

Due to the reactive nature of the SUT it is necessary to have a model of the environment. This way we can generate test sequences without actually running the SUT in its real

```
node choice () returns( x :int) =
  loop {
    | 3 : x = 42
    | 1 : x = 1
}
```

Figure 2. Lutin code, featuring a choice operator and the weights in boldfaced font, associated with the different choice possibilities.

environment. There are specialized tools for describing the environment of reactive systems. Since we are testing programs written in Lustre and SCADE, in our work, we will use Lutin [5] (a language derived from Lustre) to model the environment.

Lutin is an automatic test generator for reactive programs that focuses on functional testing. This means that the SUT will be treated as a black-box, for which we want to check some properties. Lutin enables us to perform guided random exploration of the environment, taking into account the output of the SUT, which is basically a Lustre program. This section provides a brief description of the language Lutin, focusing on the operators and non-deterministic statements of the language used to perform the guided random exploration.

The language is based on the use of descriptions of the environment, formulated in form of constraints. The constraints can be both boolean and numerical [15]. In addition, the *pre* operator enables to access the value of a given variable from the previous iteration of the system. This operator can be used in order to express temporal statements and constraints.

Lutin generates test scenarios by combining several constraints. Test input sequences for the SUT are generated by solving the constraints and randomly selecting some of the solutions.

A Lutin program is basically an automaton where each transition is associated to a set of constraints that define the possible outputs, weights that define the relative probability for each transition to be taken.

Non-determinism in Lutin is mainly realized with the non-deterministic *choice operator* |, as illustrated in the code example from Figure 2.

The weights described above enable us to influence how the environment reacts. One of the major goals of the proposed method is to optimize the weights such that the responses of the environment lead to test sequences which drive the SUT as close to safety conditions as possible. These are namely the scenarios where the malfunctioning of the SUT occurs the most often.

Besides the choice operator, non-determinism can be expressed in Lutin with *random loops*, which are defined in terms of expected number of iterations. Based on Raymond et al. [5], there are two possibilities to express the expected number of iterations:

1) $loop[min, max]$: the number of iterations should be between the constants min and max.
2) $loop \sim av : sd$: the average number of iteration should be *av*, with a standard deviation *sd*.

The parameters min, max, av, sd will be treated as subjects of the optimization process together with the above described weights of the choice operator.

C. Optimizing the environment model

In the problem of automatic test generation, the domain of possible inputs, i.e., the possible test cases is typically too large to be exhaustively explored, even for small programs. The dimensions of the search space are directly related to the number of input parameters of the SUT [7]. Since evolutionary algorithms are able to produce effective solutions for complex and poorly understood search spaces with multiple dimensions, they can also be successfully applied for testing [7][8][9][10]. However, the greatest challenge remains to formulate the testing task as an optimization problem. This will influence the success of the test case design and test input generation.

Depending on how the fitness function is formulated, evolutionary testing can be both applied for structural testing (e.g., maximizing coverage) and functional testing (e.g., fault detection).

Our approach proposed for applying an evolutionary algorithm for the optimization of the parameters of an environment model is composed of the following main steps:

1) Specifying the subject of the optimization (which parameters are to be optimized);
2) Specifying the fitness function;
3) Specifying the operators.

As mentioned in Section II-B, the language proposed for describing and optimizing the SUT environment is Lutin. In its current form, Lutin performs a guided random exploration of the SUT input state space by means of programs that describe the usage of the system [16]. The creation of these programs and the fine-tuning of their parameters however requires the domain specific knowledge of experts. To eliminate this dependency, our approach proposes to let an evolutionary algorithm choose some parameters of Lutin programs, such as those presented in Section II-B. In the first step of our approach, we need to choose the Lutin parameters that will be the subject of the optimization. Thus, the set of individuals or candidate solutions to the optimization problem will be created. This set is commonly referred to in evolutionary techniques as a *population*.

In the next step, promising individuals will be selected from the population based on a *fitness function*. Since we want to perform functional testing, we need a fitness function that measures how close the generated test cases are to violating the safety properties of the SUT and thus to detect failures in the SUT. Assertions used in Lustre to express the safety properties of the SUT (described in Section II-A) will be exploited to design the suitable fitness functions.

The third step of the optimization process is to generate a new population based on the individuals selected in the previous steps. Classical operators of the evolutionary algorithms like *mutation* and *crossover* will be used in this step.

As a result of the optimizing process, Lutin weights will drive the environment into test scenarios where the SUT will get close to violating safety properties. These scenarios will potentially cause the malfunctioning of the SUT, therefore they are the target of our optimization method.

III. Experimental validation

For evaluating the proposed method, we are carrying out experiments using simulations of a real-world, industrial problem within the domain of railway automation. The problem specification was proposed by our industrial partner, Siemens.

```
node emergencyBraking(speed:int; speedCheck,
    bac:bool)
returns (active:bool);
let
  active=false->
        if (speed >= 40) and speedCheck then
            true
        else if (speed < 40) and (not bac and
            pre(bac))
                then false
                else pre(active);
tel;
```

Figure 3. Implementation of the activation of the emergency brake in Lustre. The variable *speed* stores the speed of the train, *speedCheck* the state of the speed restriction check mode (active or inactive), while *bac* represents the button which can deactivate the brake.

The problem is related to the TBL1 system, a train protection system used in Belgium and on Hong Kong's East Rail Line. Its main role is to ensure safe operation in the case of human failure. More precisely, the TBL1 system requires the locomotive driver to manually acknowledge a warning when passing a double yellow signal, as well as stopping the train automatically if it passes a red signal. (A double yellow signal means: *Preliminary caution, the next signal is displaying a single yellow aspect*, while a Single yellow aspect signalizes the following: *Caution, be prepared to stop at the next signal*.) The system is based on a trackside beacon which sends an electromagnetic signal to an aerial located underneath the locomotive.

Besides the above mentioned ones, the TBL1 system has a speed restriction checking functionality. This feature is activated by a beacon located 300 meters up-line from a signal. If the train travels at a speed greater than 40 km/h ahead of a red signal, the TBL1 system triggers the emergency brake.

In order to run some initial experiments, we have implemented the speed restriction check functionality of the TBL1 system in Lustre. The implementation was realised based on the specification and the SCADE model of the system, provided by Siemens.

Figure 3 shows the implementation of a Lustre node responsible for the activation of the emergency brake. As already mentioned above, the brake is activated if the TBL1 system is in speed restriction check mode, and the train has a speed greater or equal to 40 km/h. The brake can be deactivated after 20 seconds manually by the driver, if the train's speed has decreased below 40 km/h. The deactivation is done by pressing and releasing the *bac* button.

To test the functionality described above, it was necessary to implement a model of the environment for which we chose the Lutin language. A part of the code is illustrated in Figure 4. Here, the Lutin code simulates the pressing and releasing of the button which can deactivate the emergency brake. It can be observed that the weight associated with the choice operators are currently hardcoded. Together with other parameters, these weights will be subject of the optimization process.

Besides the *bac* button, the speed of the train and the shape of its braking curve is also determined by the environment. The

```
node bacButton () returns (bac: bool) =
        loop {
                | 1  bac = true
                | 4  bac = false
        }
```

Figure 4. Lutin code simulating the pressing and releasing of the *bac* button. The weights in boldfaced fonts are parameters that need to be optimized.

```
node speed (emergencyBrake: bool)
returns (speed: int) =
   exist D:int [-30; 30] in
     ((speed = 0) and (D = 0))
   fby
   loop (speed = pre speed + pre D)
        and (speed>=0) and (
     if not emergencyBrake
            then ((D >= 10) and (D <= 12))
            else ((D >= -20) and  (D <= -10))
               )
```

Figure 5. Implementation of the speed function in Lutin. The initial value of the speed is 0 km/h. If the emergency brake is inactive, the speed increases with a value randomly chosen between 10 and 12; else, it decreases with a value between 10 and 20.

description of these variables is a more challenging task, since they must be calculated individually for each different train model. In our current environment model, the speed of the train is only influenced by the state of the emergency brake (active or inactive). If the brake is on, the speed decreases with a randomly selected value; otherwise it increases. Figure 5 shows the implementation of the speed function.

The SUT and environment models are connected by the Lurette tool [17]. Lurette ensures the cyclic interaction between the SUT and its environment. The values generated by the Lutin code are fed in as inputs to the SUT, while the outputs of the SUT are processed by the Lutin code. Lurette also checks the outputs generated by the SUT for some given inputs based on the test oracles, and decides whether the SUT has passed or has failed a given test case. The test oracles are also implemented in Lustre.

IV. Conclusion and Future Work

This paper presented an outline of our approach to the use of evolutionary techniques to automatically fine-tune the environment model based on which automatic test generation for reactive systems can be performed. In the design of the optimization method we made use of the choice node weights and the variable bounds from the Lutin-based environment description. In addition, we proposed to exploit Lustre assertions for measuring how close the generated test sequences get to violating the safety properties of the SUT. The outlined method could minimize the need for expert knowledge in order to model the environment and derive test cases that could find faults in the SUT. We outlined how our proposed method can be applied in a realistic simulation environment from the railway automation domain. Concrete experimental results will only be available once the implementation of the full system model and the required environment is done, based on the TBL1 specification. As part of our future work, we plan to finalize the empirical evaluation of the method and extend the proposed optimization framework to include active-learning based algorithms.

References

[1] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, “Lustre: A declarative language for real-time programming,” in Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ser. POPL ’87. New York, NY, USA: ACM, 1987, pp. 178–188.
[2] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous data flow programming language lustre,” Proceedings of the IEEE, vol. 79, no. 9, Sep 1991, pp. 1305–1320.
[3] The lustre v6 reference manual. [Online]. Available: http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf [retrieved: august, 2014]
[4] F. X. Dormoy, “Scade 6 a model based solution for safety critical software development,” ERTS 2008, 2013.
[5] P. Raymond, Y. Roux, and E. Jahier, “Lutin: A language for specifying and executing reactive scenarios.” EURASIP J. Emb. Sys., vol. 2008, 2008.
[6] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
[7] T. E. Vos et al., “Evolutionary functional black-box testing in an industrial setting,” Software Quality Control, vol. 21, no. 2, Jun. 2013, pp. 259–288.
[8] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, “Evolutionary test program induction for microprocessor design verification,” 2012 IEEE 21st Asian Test Symposium, 2002, p. 368.
[9] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose, “Automatic test program generation for pipelined processors,” in Computer-Aided Design, 1994., IEEE/ACM International Conference on, Nov 1994, pp. 580–583.
[10] A. Cheng and C.-C. Lim, “Markov modelling and parameterisation of genetic evolutionary test generations,” Journal of Global Optimization, vol. 51, 2011, pp. 743–751.
[11] L. d. Bousquet and N. Zuanon, “An overview of lutess: A specification-based tool for testing synchronous software,” in Proceedings of the 14th IEEE International Conference on Automated Software Engineering, ser. ASE ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 208–215.
[12] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon, “Lutess: A specification-driven testing environment for synchronous software,” in Proceedings of the 21st International Conference on Software Engineering, ser. ICSE ’99. New York, NY, USA: ACM, 1999, pp. 267–276.
[13] B. Marre and A. Arnould, “Test sequences generation from lustre descriptions: Gatel,” in Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ser. ASE ’00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 229–237.
[14] B. Marre and B. Blanc, “Test selection strategies for lustre descriptions in gatel,” Electronic Notes in Theoretical Computer Science, vol. 111, Jan 2005, pp. 93–111.
[15] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber, “Automatic testing of reactive systems,” in Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE, Dec 1998, pp. 200–209.
[16] E. Jahier, S. Djoko-Djoko, C. Maiza, and E. Lafont, “Environment-model based testing of control systems: Case studies,” in Tools and Algorithms for the Construction and Analysis of Systems, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds. Springer Berlin Heidelberg, 2014, vol. 8413, pp. 636–650.
[17] E. Jahier, P. Raymond, and P. Baufreton, “Case studies with lurette v2,” Int. J. Softw. Tools Technol. Transf., vol. 8, no. 6, Oct. 2006, pp. 517–530.

Inconsistencies-based Multi-Region Protocol Verification

Tukaram Muske, Amey Zare
TRDDC, Tata Consultancy Services,
54 B, Hadapsar I.E., Pune, India
{t.muske, amey.zare}@tcs.com

***Abstract*—Software in complex systems like embedded systems usually include protocols (Sleep Wakeup, Controller Area Network Communication, and so on) implemented in multiple code-regions, and these protocols are crucial for the system correctness. For such protocol implementations, code review and testing often fail to detect some of the critical bugs. Many of these bugs are traced back to inconsistencies in the implemented code-regions. We present a new verification technique that identifies likely coding inconsistencies by computing and comparing protocol-critical information over given protocol code-regions. These inconsistencies are then manually validated. In our experiments, the presented technique detected critical bugs that were missed during code reviews and testing.**

***Index Terms*—Embedded Systems; Validation and Verification; Protocols Verification; Coding Inconsistencies**

I. Introduction

Complex systems such as embedded systems usually implement various protocols like as Security, Controller Area Network (CAN) communication and Sleep Wakeup in certain patterns. In one pattern, actions of these protocols are implemented over several parts of the system. For example, actions in CAN communication protocol [1] include data encoding/decoding, message and acknowledgments sending/receiving, error checking, and so on. These actions are often complementary to each other and are implemented in different parts of the code. Further, the actions in a protocol may be required and implemented by multiple components of the embedded system, leading to several similar implementations of the protocol. Henceforth in the paper, a part of code that implements protocol action(s) separately is referred to as a *region*. Thus, such an implemented protocol consists of multiple code regions, and functionality wise, they can be similar or opposite.

Verification of an embedded system is of utmost importance [2]–[4], and it includes proving correctness of intended system functionalities and making sure unintended behaviors are absent [5]. In order to do this, it has become necessary to ensure the included protocols are correctly implemented, as they are crucial for correctness of the system functionality. We describe this by using a Sleep Wakeup protocol that is usually implemented in three regions - Startup, Sleep and Wakeup [6]. This protocol is typically used in battery-powered systems to minimize power consumed by a microcontroller (also referred to as Electronic Control Unit (ECU)). Such power minimization is achieved by toggling the ECU between low power consumption mode (*Sleep* state) and high power consumption mode (*Run* state).

Figure 1 presents a sample implementation of Sleep Wakeup protocol. In this implementation, the Sleep region starts at line 41 and ends at line 45. This region performs certain actions to reduce power consumption before an ECU enters the *Sleep* state. These actions include configuring registers (hardware ports), disabling hardware such as timers, CAN communication channels, and ADC (Analog to Digital Converter). The Startup region (lines 11 to 18) and Wakeup region (lines 83 to 90) perform similar actions before the ECU enters the *Run* state. These actions are opposite to the actions performed in the Sleep region and often include reconfiguring the registers, enabling hardware that were disabled before entering the *Sleep* state.

When a protocol is implemented in multiple regions, possibility of defect introduction increases. This is explained below with respect to the sample implementation.

1) Let us assume that the system functionality (implemented by *perform_Job* function) requires register TRISA (Port A) to be configured with 0x00 value. The Startup region performs the expected register configuration, however the Wakeup region misses to do so. This mismatch can lead to unexpected system behaviors.
2) The CAN communication channel (CANCHNL0) and timer (Timer0) are always enabled in the Startup and Wakeup regions, however their disabling is missed in the Sleep region. Having them enabled during the ECU *Sleep* state can result in more power consumption, and this may lead to battery drain.
3) The hardware HWx is always turned *off* in the Sleep region, but it is missed to turn *on* in the Wakeup region. Due to this, it remains *off* during execution of the system functionality code, and a read or write access to such hardware can result in unexpected *watch dog timer reset* [7].
4) The analog-to-digital converter (ADC) is always disabled in the Sleep region, whereas it is possibly enabled in the Wakeup region. When the Wakeup region fails to enable the ADC, accessing the ADC in the system functionality code can lead to unexpected behaviors.

The above described defects are introduced due to coding inconsistencies, and detecting all such defects in practice

```
 1. int main()
 2. {
     ...
11.  // Startup region begin
12.  TRISA = 0x00; //Port A configuration
13.  CANCHNL0=0x0010; //Enable CAN channel
14.  Timer0 &= 0x80; //Enable Timer 0
15.  HWx = 0x0000; //Switch-on hardware X
16.  ADC = 0x0010; //Enable ADC
17.  var1 = 10;
18.  // Startup region end
19.  ...
20.  while(1)
21.  {
22.   perform_Job();//System functionality
23.   sleep_Wakeup_sequence();
24.  }
25. }
```

```
31. void sleep_Wakeup_sequence(){
     ...
41.  // Sleep region begin
42.  TRISA = 0xff; //Port A configuration
43.  HWx = 0xffff; //Switch-off hardware X
44.  ADC = 0x0001; //Disable ADC
45.  // Sleep region end
46.  SYS_REG = 0x7fff;//ECU in Sleep state
47.  ...
48.  while( !wakeup_condition );
     ...
81.  SYS_REG = 0x8fff; //ECU in Run state
82.  ...
83.  // Wakeup region begin
84.  TRISA = 0xff; //Port A configuration
85.  CANCHNL0=0x0010;//Enable CAN channel
86.  Timer0 &= 0x80; //Enable Timer 0
87.  if(...){
88.    ADC = 0x0010; //Enable ADC
89.  }
90.  // Wakeup region end
91. }
```

Fig. 1. Sample implementation of a Sleep Wakeup protocol

through code reviews may not be possible. This is because, the protocol regions can start and end anywhere in the application and may span over thousands of lines of code that configures (initializes) hundreds of registers (variables). Further, it is not always guaranteed that all such defects will be detected during system testing. For example, the defect (2), due to miss of disabling of CAN channels and timers in the Sleep region, can not be observed via system output parameters. Hence, detecting this defect using testing is difficult as it requires use of sophisticated power consumption monitoring techniques. Due to these issues, a verification technique that helps in automatic and early detection of such defects is always useful.

This paper presents a verification technique that accepts the protocol regions to be verified as inputs and detects possible defects by checking consistency over these regions. This technique is based on computing certain protocol-critical information over each of these regions (referred to as Regional Information), comparing the regional information, and raising an inconsistency so found as a possible defect. Further, this paper shortly describes a framework to compute the required regional information over the input regions. The described framework, first identifies program points that lie inside a given region and later computes the regional information as an effect of the identified program points. This technique to compute regional information is referred as *regional analysis*.

We applied the proposed inconsistencies detection technique to verify Sleep Wakeup protocols in two C applications from automotive industry. The empirical results indicate - a) the presented verification technique detects defects in protocol implementations, which are missed by the other defect finding techniques such as testing and manual code reviews, *and* b) like for any other static analysis technique, generation of false alarms is a concern for our technique.

The key contributions of this paper are - a) an idea to break a complex protocol implementation into similar or opposite regions for the protocol verification, b) a framework for the regional information computation, *and* c) an approach to detect likely inconsistencies by comparing regional information and viewing them as possible defects.

Paper outline: Section II describes the inconsistencies-based verification technique, and Section III provides details of the regional analysis framework. The experiments and their results are described in Section IV. Section V and Section VI, respectively, present related work and conclusion.

II. Multi-Region Protocol Verification

This section describes an inconsistencies-based approach to verify a multi-region protocol implementation by using examples of Sleep Wakeup and CAN communication protocols.

A. Protocol Region: Definition

We define a region starting at P_S and ending at P_E as the part of code having program points that appear on a path originating at P_S and terminating at P_E. The program points in a given region, thus identified, are referred to as *in-region* points, and they include assignment and conditional statements, calls to functions, return statements, etc. Table I provides a few sample regions for a code snippet shown in Figure 2 and their in-region points excluding the region boundaries. It uses line numbers to denote the program points.

As there exists a variety of protocols and each protocol can be implemented in different ways, automatic identification of regions is difficult and may not be generic. Hence, we accept them as inputs specified by their start and end points. In practice, the start and end points of a given region can appear anywhere in the application, and only the code belonging to the given region needs to be analyzed for computation of the intended regional information. For example, in Figure 1, variable ADC is *possibly* modified over the Wakeup region (lines 83 to 90), but computing this modification type over

```
 1. void main()            | 21. void func()
 2. {                      | 22. {
 3.    gVar1 = 1;          | 23.    var1 = 1;
 4.    while(c1)           | 24.    if ( c2 )
 5.    {                   | 25.       var2 = 2;
 6.        gVar2 = 2;      | 26.    else
 7.        func();         | 27.       var3 = 3;
 8.        gVar3 = 3;      | 28.    var4 = 4;
 9.        gVar4 = 4;      | 29.    if ( c3 )
10.    }                   | 30.       var5 = 5;
11.}                       | 31. }
```

Fig. 2. Code snippet for sample regions

TABLE I
SAMPLE REGIONS

Sample Region	Start Point	End Point	In-Region Points
I	6	8	7, [23-30]
II	6	28	7, [23-27]
III	28	6	29, 30, 8, 9, 4
IV	28	23	29, 30, 8, 9, 4, 6, 7
V	25	9	[28-30], 8
VI	25	30	[27-29], [4-9], 23, 24
VII	30	25	[4-9], 23, 24, [27-29]

the complete function (*sleep_Wakeup_sequence*) would find it as *definite*.

B. Sleep Wakeup Protocol Verification

As discussed earlier in Section I, the defects in the sample protocol implementation in Figure 1 arise due to either wrong or miss of a register/variable initialization in one of the regions. In order to detect these defects, a systematic approach is required to select, compute, and compare suitable information over the protocol regions.

1) Regional information computation: We select and compute below regional information to verify a Sleep Wakeup protocol implementation.

i. *Regional Modification Type:* Modification type of a register/variable over the protocol regions is suitable to check if these regions consistently modify the involved register/variable. An inconsistency found this way represents a miss of a register/variable initialization in one of the regions. Computation of such regional modification type of a variable v over an input region starting at R_S and ending at R_E points is described below.
   a) *Definite (D):* The regional modification type of v is *definite* if each path originating at R_S and ending at R_E modifies v.
   b) *Possible (P):* The regional modification type of v is *possible* only if v is modified along at least one path and not by all the paths that originate at R_S and end at R_E.
   c) *No (N):* v has *no* modification type when no path originating at R_S and ending at R_E modifies v.

ii. *Regional Values:* Values assigned to a variable/register over each of the input regions (regional values) are suitable to detect mismatch in the initialization values of the register/variable. Such a mismatch in the regional values over two expected similar regions represents a wrong initialization of the register/variable in any one of the regions.

2) Regional inconsistencies detection: In order to detect the possible defects, the regional information computed over each of the Sleep Wakeup protocol regions is compared as described below.

Comparing Regional Modification Types: Regional modification types of a register/variable are compared in below combinations to detect miss of a register configuration (variable initialization).

a) Startup Vs Wakeup, because a register configuration (variable initialization) in the Startup region should have its corresponding configuration (initialization) in the Wakeup region.
b) Sleep Vs Wakeup, because a register configuration in Sleep region should have its opposite configuration in the Wakeup region. It is to note that, only registers are considered in this combination, and the hardware ports, timers, ADC, other hardware, etc are to be treated as the registers.

Given the comparisons in the aforementioned combinations, it is intuitive that such comparisons are not required for Startup Vs Sleep. In these comparisons, consistency is reported only if both the modification types being compared are *definite*, and all other comparison scenarios are treated as inconsistencies. To be conservative, comparison of two *possible* modification types is treated as an inconsistency, since in this setting, a configuration in one region can not be guaranteed to have its corresponding configuration in the other region. Such a conservative approach may lead to an increased number of inconsistencies and thus a high rate of false alarms.

Comparing Regional Values: Regional values of a register/-variable are compared to detect a wrong configuration/initialization. When the regional values over one region differ with the values from some other similar region, such a scenario is reported as an inconsistency. Also, when the regional values of a register/variable can not be computed statically or are found as interval of values, to be conservative, their comparisons are reported as inconsistencies. Such regional value comparisons are performed only for the Startup Vs Wakeup combination. The Sleep Vs Wakeup combination is considered since the variable/register values are not expected to be same over these regions.

Table II presents results of the consistency checks performed for the protocol implementation in Figure 1. All the defects in the implementation (as described in Section I) are represented by the inconsistencies shown in this table.

The regional information used to detect likely inconsistencies is not limited to modification type and values. One can use other Sleep Wakeup protocol-critical information such as call type (*definite, possible and no*) of instructions that enable/disable the interrupts, and other system calls that acquire and release the locks. Comparing such regional information can help to identify miss on a call of these system calls or instructions.

TABLE II
CONSISTENCY CHECKS OVER SAMPLE REGIONS

Variable /Register	Regional Modification Type			Regional Values			Regional Modification Type		
	Startup	Wakeup	Consistency?	Startup	Wakeup	Consistency?	Sleep	Wakeup	Consistency?
TRISA	D	D	$\surd$	0x00	0xff	X	D	D	$\surd$
CANCHNL0	D	D	$\surd$	0x0010	0x0010	$\surd$	N	D	X
Timer0	D	D	$\surd$	0x80	0x80	$\surd$	N	D	X
HWx	D	N	X	0x0000	-	NA	D	N	X
ADC	D	P	X	0x0010	0x0010	$\surd$	D	P	X
var1	D	N	X	10	-	X	N	N	NA

C. CAN Communication Protocol Verification

Selection of the regional information for inconsistencies detection varies as per the protocol. For example, the regional information used to verify an implementation of CAN communication protocol may not be same as it is used in the Sleep Wakeup protocol verification. This is because, a CAN protocol is usually implemented by several components of an automobile embedded system such as *wiper, flasher*, and *body* control unit. Thus, there are repeatative implementations of the same CAN protocol and they ought to be consistent with each other.

The regional information that can be used to verify such implementations may include - a) call type (*definite, possible and no*) of the communication services/APIs [1], b) calling sequence of above services/APIs, c) modification type of parameters of the services/APIs related to message sending, d) read type of parameters of the services/APIs related to message receiving.

It is to note that the presented protocols verification technique is not limited to Sleep Wakeup and CAN communication protocols. Through suitable regional information identified, this technique can be applied to other protocols whose implementation is distributed over multiple similar or opposite regions.

III. REGIONAL ANALYSIS FRAMEWORK

This section briefly describes a framework to compute regional information over a region specified by its start and end points. This computation is achieved in two steps. In the first step, in-region points for a given region are marked (referred to as *region marking*), and in the next step, the in-region points are analyzed to obtain the required regional information.

A. Region Marking

As discussed earlier (in Section II-A), identification and analysis of in-region points is essential to compute the intended regional information. Although, we have defined the in-region points by referring to paths between the region boundaries, computing them this way may not be feasible in practice since the number of paths grows exponentially to number of the conditions. Thus, we use *may* and must reachabilities of a program point from region boundaries, in forward and backward direction, to identify if the program point is an in-region point.

Definition: In-region point- A program point P is an in-region point with respect to a region having R_S and R_E as its start and end points respectively only if both of the following hold true.

1) In forward flow, P is *may* reachable from R_S and it is not *must* reachable from R_E.
2) In backward flow, P is *may* reachable from R_E and it is not *must* reachable from R_S.

Here, the *may* reachability subsumes the *must* reachability. Due to space constraints we avoid detailing the region marking step further.

B. Regional Information Computation

The regional information used in inconsistencies detection can be of several types and varies as per the protocols being verified. The regional modification types and values of the variables are applicable to most of the protocols. Thus, we describe their computation as a representative example of the regional information computation.

1) Computation of regional modification types: Data flow analysis [8] is suitable to compute the *must* and *may* modified variables, and their corresponding data flow formalizations are shown in Table III. For simplicity of the shown formalizations, we have assumed the region code is free of pointers and the region boundaries lie in the same function. These formalizations use results of the region marking to compute the required regional information over the in-region points only. In these formalizations, In_n represents the information flowing in at the start of a node n, while Out_n represents the information flowing out of the exit of the node n. The Gen_n corresponds to the information generated as an effect of the node n.

Using *must* and *may* modified variables over a given region, the regional modification types of the variables can be obtained. The *must* modified variables have the *Definite* modification type. The variables which are *may* but not *must* modified, have *Possible* modification type. A variable that is not *may* modified, has *No* modification type.

2) Computation of regional values: A data flow formalization similar to the formalizations shown in Table III can be used to compute the regional values. Due to space constraints, we avoid providing a separate formalization for regional values computation.

IV. EXPERIMENTAL RESULTS

This section describes various experiments performed to verify the Sleep Wakeup protocols and observations from the results.

TABLE III
DFA FORMALIZATIONS FOR REGIONAL MODIFICATION TYPES

Parameter	***Must* Modified Variables**	***May* Modified Variables**
Initialization (Top)	Set of all variables in the application	$\emptyset$
Meet/Join	Intersection	Union
$In_n =$	$\begin{cases} \emptyset & n \text{ is start of a function} \\ \bigcap\limits_{p \epsilon pred(n)} Out_p & \text{Otherwise} \end{cases}$	$\begin{cases} \emptyset & n \text{ is start of a function} \\ \bigcup\limits_{p \epsilon pred(n)} Out_p & \text{Otherwise} \end{cases}$
$Out_n =$	$In_n + Gen_n$	$In_n + Gen_n$
$Gen_n =$	$\begin{cases} Top & n \text{ is an } \textit{out-region} \text{ point} \\ v & n \text{ is an } \textit{in-region} \text{ point and defines } v \\ \emptyset & \text{Otherwise} \end{cases}$	$\begin{cases} v & n \text{ is an } \textit{in-region} \text{ point and defines } v \\ \emptyset & \text{Otherwise} \end{cases}$

We implemented the described Sleep Wakeup protocol verification technique in TCS Embedded Code Analyzer (TCS ECA) [9]. TCS ECA is a static analysis tool to verify C source code. We selected two C applications from automotive industry, one of 56 KLOC representing automobile Body Control Module (BDCM) and another of 40 KLOC representing automobile Battery Control Module (BTCM). The boundaries of the Startup, Sleep, and Wakeup regions from both the applications were provided as inputs to TCS ECA during verification of the protocols. Table IV presents information about each of the regions from the selected protocols. This information includes size of the region, number of variables with the *Definite* modification types (DMTVs), and number of variables with the *Possible* modification type (PMTVs).

TABLE IV
REGIONAL ANALYSIS RESULTS

Application	Region	LOC	DMTVs	PMTVs
BDCM	Startup	3168	302	283
	Sleep	838	32	65
	Wakeup	3193	299	288
BTCM	Startup	2246	59	150
	Sleep	1150	62	18
	Wakeup	2246	59	150

TABLE V
COUNTS OF REGIONAL (IN)CONSISTENCIES

Application	Startup Vs Wakeup				Sleep Vs Wakeup	
	RMTIs	RMTCs	RVIs	RVCs	Register RMTIs	Register RMTCs
BDCM	413	186	143	544	66	33
BTCM	150	59	48	161	48	32

A. Observations from protocols verification

Table V presents the summary of the verification results of the selected Sleep Wakeup protocols. In this table, RMTCs (RMTIs) denotes count of the regional modification type consistencies (inconsistencies), and the RVCs (RVIs) denotes count of the regional values consistencies (inconsistencies).

BDCM Application: Large number of inconsistencies were reported for this application due to *possible* modification types, and its reason was traced to the conditional calls of the functions that initialized the registers/variables in the Wakeup region. Manual review of all the reported inconsistencies, performed by the system developers, took around two hours of manual efforts. Few observations from this activity are mentioned below.

- The review of the RMTIs in Startup Vs Wakeup combination revealed *possible* miss of configuration of four hardware pins in the Startup region. This was due to conditional call of the function that configured the hardware pins. Also, review of these inconsistencies indicated that initializations to two global variables were *definitely* missed in the Wakeup region, and each miss was found to be a coding defect.
- One inconsistency among the reported 86 register RMTIs in Sleep Vs Wakeup combination indicated presence of critical defect, which was due to miss of disabling of the DMA controller in the Sleep region.
- None of the regional values inconsistency in Startup Vs Wakeup represented a coding defect.

BTCM Application: The Startup and Wakeup regions in BTCM were found to be overlapping, hence the inconsistencies reported for this combination were not manually reviewed. On manual review, none of the register modification inconsistency in Sleep Vs Wakeup represented a coding defect. This manual review took around 20 minutes.

B. Other Observations

Inconsistencies-based verification Vs Manual Code Review: We performed an experiment to check effectiveness of the presented verification technique against the manual code review. In this experiment, a developer manually reviewed the protocol code in BDCM application to identify the defects. After six hours of reviewing efforts, the developer was able to identify only one defect related to the miss of disabling of DMA controller, and the other defects were not found during the review. This experiment indicated the presented technique is useful in detecting more bugs which are usually missed during code reviews.

Inconsistencies-based verification Vs Testing: Both the selected applications were after their testing at unit and system levels. The testing of the selected BDCM application was unable to detect the defects that were detected by the inconsistencies-based verification approach. It indicated the effectiveness of the presented verification technique in detecting defects which are hard to find during testing.

Impact of conservative approach: We performed few experiments to observe impact of the conservative approach taken during the computation of inconsistencies. These experiments indicated that around 55% of the reported inconsistencies were due to treating a comparison of two *possible* modification types as an inconsistency. In our experiments, although the inconsistencies due to conservative approach did not contribute in defects identification, we believe such an approach may benefit on some other applications. Further, these experiments indicate that the conservative approach can be avoided in order to generate fewer false alarms at the cost of miss of detection some defects.

V. Related Work

Coding inconsistencies have been used earlier for bugs detection. Engler et al. [10] used automated rule extraction to get the programmer beliefs, and one of the contradictory beliefs are treated as an error. Lu et al. [11] have used an inconsistency in updates to the correlated variables for semantic bugs detection. To the best of our knowledge, such coding inconsistencies detection has not been used in the verification of protocol implementations. In our presented technique, the information with which the inconsistencies are computed is critical to the protocol functionality, and it is based on the (dis)similarity of the actions implemented by protocol regions. This regional information is different from the information used by the existing techniques [10][11].

There are a number of protocols corresponding to security, communications, cryptography (data encryption), routing in networks, etc, and many approaches have been proposed to verify their implementations. These approaches use a variety of techniques such as predicate abstraction [12], patterns-based verification [13], model checking, heuristic search, or their combinations [14]. The approaches used and categories of the bugs detected by these techniques are protocol-specific. None of these techniques break a complex protocol implementation into the similar or opposite functionality regions and achieve the protocol verification.

Regional analysis has been used mostly earlier for effective memory management [15][16] and efficient solving of the data flow analysis [17], but it has been rarely used in protocol verifications. In these existing techniques, the regions to be analyzed are automatically identified, where the region boundaries belonged to the same function. Our presented regional analysis framework analyzes a given region whose boundaries can appear anywhere in the application.

VI. Conclusion and Future Work

An idea to break a complex implementation of a protocol into similar or opposite regions and an approach for their verification was presented in this paper. Detecting inconsistencies over the multiple regions of a protocol is an effective verification technique, since there could be a mismatch in their implementations due to coding by multiple developers and its multi-place distribution. Further, discovering such an inconsistency and its associated defect may not be easy using manual reviews and/or conventional testing techniques. Similar have been our observations during the experiments, which indicated usefulness of the presented technique in detection of the critical defects.

The thorough manual review of the reported inconsistencies increased our confidence about correctness of the protocol implementation. This process acted as a systematic review of the implementations, which would have not been possible otherwise. Although the experiments are performed on Sleep Wakeup protocols in embedded domain applications coded in C, we expect similar benefits on other domain/language protocols too, due to common coding practices.

Like any other static analysis technique, our experiments depicted a very high rate of false alarms (around 98%) for the presented verification technique. We plan to work on minimizing falsely reported inconsistencies in the near future.

References

[1] AUTOSAR, "Autosar specification of communication v2.0.1," Jun. 2006.

[2] J. C. Knight, "Safety critical systems: challenges and directions," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE, 2002, pp. 547–550.

[3] D. R. Wallace and R. U. Fujii, "Software verification and validation: an overview," *Software, IEEE*, no. 3, pp. 10–17, 1989.

[4] J. Yoo, E. Jee, and S. Cha, "Formal modeling and verification of safety-critical software," *Software, IEEE*, no. 3, pp. 42–49, 2009.

[5] P. Cousot and R. Cousot, "Verification of embedded software: Problems and perspectives," in *Embedded Software*. Springer, 2001, pp. 97–113.

[6] AUTOSAR, "Autosar specification of ecu state manager v3.0.0 r4.0 rev 3," Nov. 2011.

[7] J. Santic, "Watchdog timer techniques," *Embedded Systems Programming*, vol. 8, no. 4, pp. 58–69, 1995.

[8] U. Khedker, A. Sanyal, and B. Sathe, *Data Flow Analysis: Theory and Practice*. Taylor & Francis, 2009.

[9] TCS Embedded Code Analyzer (TCS ECA), http://www.tcs.com/offerings/engineering_services/Pages/TCS-Embedded-Code-Analyzer.aspx, [Online; accessed 25-Aug-2014].

[10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, Oct. 2001.

[11] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 103–116, Oct. 2007.

[12] E. Pek and N. Bogunovic, "Predicate abstraction in protocol verification," in *Telecommunications, 2005. ConTEL 2005. Proceedings of the 8th International Conference on*, vol. 2. IEEE, 2005, pp. 627–632.

[13] L. Bozga, Y. Lakhnech, and M. Périn, "Pattern-based abstraction for verifying secrecy in protocols," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 299–314.

[14] S. Edelkamp, A. L. Lafuente, and S. Leue, *Protocol verification with heuristic search*. Bibliothek der Universität Konstanz, 2001.

[15] S. Cherem and R. Rugina, "Region analysis and transformation for java programs," in *Proceedings of the 4th international symposium on Memory management*. ACM, 2004, pp. 85–96.

[16] R. Rugina, "Region analysis for imperative languages," Cornell University, Tech. Rep., 2003.

[17] Y.-F. Lee, B. G. Ryder, and M. E. Fiuczynski, "Region analysis: A parallel elimination method for data flow analysis," *Software Engineering, IEEE Transactions on*, vol. 21, no. 11, pp. 913–926, 1995.

Towards a Holistic Architecture for a SIP Test Framework

Teemu Kanstrén, Pekka Aho
VTT, Oulu, Finland
teemu.kanstren@vtt.fi

Abstract— **Testing traditionally focuses on specific aspects of a system separately, such as functional conformance, robustness and performance. In this paper, we present a protocol test automation framework that considers these different viewpoints as a whole. It starts with a common architecture for protocol testing at different scales. A set of test models on top of this architecture are presented for addressing the different types of testing. Each of these models builds on top of another, starting with conformance testing, followed by robustness testing and finally overall performance testing. We apply the framework to session initiation protocol (SIP).**

Keywords- Test automation, framework, sip, protocol testing

I. INTRODUCTION

Testing is a multifaceted discipline. It needs to verify various aspects of system behavior, including performance, robustness, and functional conformance. Different types of testing further have various coverage criteria, many specific to the type of testing and to the domain of the system under test (SUT). Covering these different types of testing and their different coverage criteria extensively can be very expensive. Commonly each system is also different, and creating largely re-usable test automation frameworks and test suites is difficult. In such cases, we commonly choose the most critical pieces of the SUT and target most of our coverage on those parts.

When systems are based on a set of standardized protocols, the test frameworks and test suites for those parts can be more extensively re-used and potential for extensive test automation frameworks is higher. A test framework for a standardized protocol can be applied on many different systems. In fact, protocol testing is an active field of research and various tools for testing different aspects of different protocols exist. A large scale example of this is the protocol conformance testing effort by Microsoft for testing more than 250 protocols [1]. For robustness testing, another example is protocol fuzzers which are a popular type of tool used to test robustness protocol implementations [2] and a popular research topic (e.g., [3, 4]). Fuzzers manipulate the protocol messages and the contained data to evaluate how the implementation can handle malformed inputs.

However, while there exist a wide range of protocol testing research and tools, these traditionally target a narrow part of the overall quality assurance for a protocol and the system built using that protocol. In this paper, we present a holistic test automation framework for protocol testing. It is aimed at supporting testing the protocol implementation at the basic protocol stack level, testing the protocol application to communication between two nodes, and to testing the application of the protocol to the overall communication in a large distributed system. It is also aimed at supporting conformance testing, robustness testing, and performance testing using model-based techniques with each testing type building on top of the previous one.

While some adaptation of the framework architecture is required in going from testing a protocol stack in isolation to testing the protocol use at larger scale, defining the overall common concepts enables us to build reusable components for the overall framework and to systematically build better quality into the different layers. With the different model-based test approaches we gain a diverse coverage for different types of testing, while reducing the costs in building the different models as layers on top of each other.

As our work on this has been practically performed in the context of the session initiation protocol (SIP), we present the application of the framework and its specialization for SIP as a running example throughout the paper.

The rest of the paper is structured as follows. Section II introduces the important background concepts and related works. Section III presents the components of our framework architecture. Section IV presents the set of test models for different types of testing and their composition. Section V discusses the concepts more broadly, and finally conclusions end the paper.

II. BACKGROUND AND RELATED WORKS

In this section, we give a brief introduction to relevant concepts for this paper, and present related works in protocol testing.

A. Session Initiation Protocol (SIP)

Throughout the rest of this paper, we will use session initiation protocol (SIP) as a running example to demonstrate the relevant concepts. SIP is a protocol used for signaling in setting up communication sessions. Typical usage scenario is Voice over IP (VoIP), where different endpoints use SIP based communications to signal call control flow. Various other protocols can then be used for the actual call (such as voice and video transport). We focus here only on control flow signaling which is what the SIP protocol is for. Figure 1 illustrates the basic call flow in such scenario.

Beyond this basic call setup shown in Figure 1, there can be various configurations such as the call going through one or more proxies, changing call details in mid-call (e.g., re-negotiating quality parameters), or several parties together negotiating a conference call. Figure 2 illustrates these different configurations from the testing perspective in four different cases (A-D).

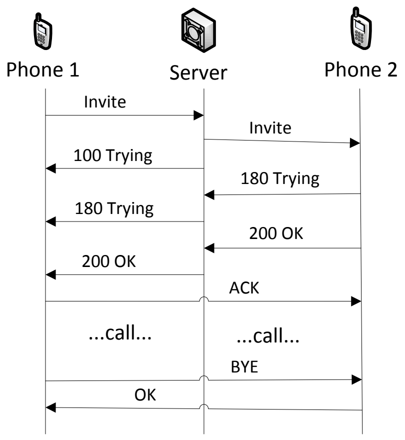


Figure 1. Basic SIP call flow.

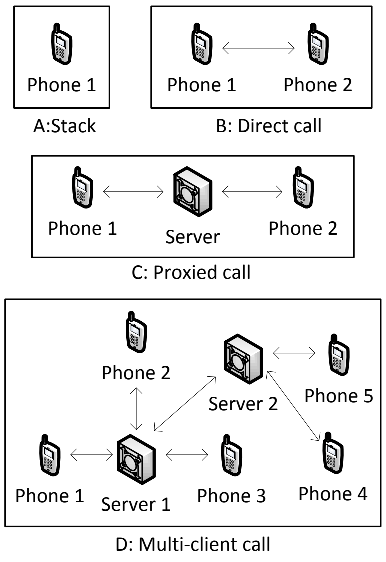


Figure 2. SIP system configurations.

In the rest of the paper, we will refer to these different scenarios in Figure 2 as scenarios A, B, C and D, or more generally scenarios A-D. In scenario A, we test the protocol stack in isolation as a single unit (or module). In scenario B, we test two devices communicating directly with each other. In scenario C, we test two devices communicating with a server connecting them. In scenario D, we have multiple devices all communicating together in a single call session, potentially across several servers as well. Sometimes these clients can also move during the call between servers (mobile clients). We will look at testing these in more detail in Section III with our test framework architecture and in Section IV with our test models.

B. Model-Based Testing

Model-based testing (MBT) is a concept we use widely in this paper. We follow the definition of MBT given in [5] as "generation of test cases with oracles from a behavioral model". That is, the system is described using a behavioral model, in our case as a set of rules and actions, and test cases to exercise the behavior of interest are generated from these models by a test generator tool. SIP conformance testing is a case study quite often used in the MBT literature [5]. However, as we discuss in Section II.C, the existing work is mainly limited to conformance testing and in this paper, we discuss this more broadly with also applications to non-functional testing of robustness and performance.

As mentioned, our use of MBT is based on test models defining a set of rules and actions. In this case, the actions define some functions to be executed on the SUT. In the case of SIP these actions are typically sending SIP request messages. The rules in the test model define when each of these actions are allowed. Following these rules, a test generator can then produce a set of test cases following the test specification (the test model). As the generator follows the rules, it produces valid test cases. These are valid from the test model perspective, and thus we can also define, for example, a robustness test model for producing invalid data and invalid sequences. In such a case, the test model will describe the types of invalid data we are interested in.

In addition to the SIP requests in the test model, SIP responses are handled by the overall test framework as we will discuss in Section III. All these elements are linked to form the overall test framework, including test oracles at chosen detail level.

Figure **3** shows some example rules and actions for testing a single device SIP scenario. The solid boxes are the actions and the attached dashed boxes are the rules for those actions.

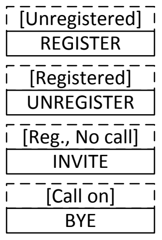


Figure 3. Example rules and actions.

In this case, the names in the action boxes in Figure **3** are different SIP request messages. For example, *invite* is a message used in SIP to establish a call between two parties. However, if a SIP proxy server is used, the parties (SIP user agents) have to *register* with the SIP server to allow the call (the "registered" rule in Figure 3). This is only allowed if not yet registered, and the registration action updates the model state to registered. Once registered, *invite* is then allowed. After a call is established, it can be terminated using the *bye* message.

The model in our case is a form of a *model program*, implemented to send the SIP messages for these actions and to maintain the state of the test client. This state is then used in the rules to define when actions are allowed. This is a common approach to MBT as described in e.g. [5, 1].

In addition to sending request messages, the tester needs to be able to handle other protocol messages as well. For example, the *Trying*, *OK* and *ACK* messages shown in Figure **1** cannot be expressed as actions as they are not actions initiated by the test client (or the test model). Instead, the test framework must be able to process such received messages from the test target(s), update its state (e.g., call established) and to check that required responses are received correctly (such as *Trying* message when expected). That is, the basic functionality of the protocol should be a part of the test framework and the test model is built on top of this. Verification of this test framework then comes from its application to several test targets, each providing a verification of the applied test framework itself.

C. SIP Protocol Testing

One the largest efforts in protocol testing is the Microsoft protocol documentation assurance effort described in [1] for testing more than 250 different protocols to ensure documentation quality and regulatory conformance. The process and tools to perform this validation have been described in detail in [1]. Conformance testing for a protocol includes describing the expected normative behavior of the protocol, meaning the allowed and required messages, their sequences and the data values. In the Microsoft case, both model-based and manual test creation methods were used, which in our experience reflects the general good test automation practices. In such a case, a test model reflects the protocol behavior, and test cases can be generated with a MBT tool from this model. Test execution is built on top of a component based adapter layer, which also forms a basis for creating manual test cases as required. Our general test automation architecture adapts elements from this and includes addressing also larger scale distributed systems and different robustness and performance testing. It thus also enables more complex and realistic test scenarios.

An example test automation framework for SIP conformance testing is presented in [6]. In this case, the framework provides a simulated SIP service environment that can be configured to provide different responses and services for SIP user agents. For example, an emulated user agent can be configured to perform call forwarding for a specific user, allowing the tester to focus on scenarios that make use of such SIP services. The work in [6] is aimed at conformance testing of different services built on top of SIP, and non-functional testing (performance) is left out of scope.

A test automation framework for performance testing is presented in [7]. This framework uses existing SIP platforms and tools such as SailFin [8] and SIPp [9] to generate different types of traffic to test performance of SIP agents. This includes traffic bursts, linearly increasing traffic and other such usage profiles. Mentioned problems include difficulty to implement complex interactions between agents as well as control of generated traffic due to limitations of the third party tools used for generating traffic. That is, the external tools used do not have support for the required level of control in fine grained performance testing. We use similar traffic profiles as part of our performance models, and integrate these with conformance and robustness test models.

A test automation approach based on passive monitoring of operational SIP based systems is presented in [10]. In this case, the idea is to describe the system expected behavior as a set of formalized properties, and use operational monitoring to assess whether the observed system behavior matches this expected specification. A similar approach taken in [11] provides a general specification of a protocol, creating a model of system behavior in different phases starting from observing network traffic to grouping it as transactions and dialogs. Finally, these are compared to the set of rules given in the specification. We do not explicitly do this type of passive monitoring as we also perform active generation of request messages. However, the part of our test framework related to listening for response messages and asserting the overall system behaviour based on those responses and their relation to test model state uses similar concepts.

During performance testing, we have to collect various metrics on system performance to evaluate how the different actions and parameters applied affect the performance. These measurements are collected by deployed probes, which can be located on the different nodes in a distributed system, or measuring the connecting network. For example, [12] describes using numerous measures collected such as CPU load, network load, interrupts and context switches on SUT nodes as a basis for a performance model. These are combined to form a detailed view of how the different components in the system affect the system performance. A high-level, specific, metric for performance testing of networked services is suggested in [13] as throughput. Throughput here means the number of requests (or transactions) performed on the system over a given time unit (such as a minute). In summary, we need both a high-level definition of what system overall performance means for us, as well as means to find the bottlenecks when relevant. In our case, we use the number of SIP messages processed as a basic metric, and focus using more detailed probes where necessary.

For robustness testing, a stateful fuzzer for SIP is presented in [3]. This is based on two components: syntax fuzzer and state evaluator. Besides the traditional data fuzzing and checking of aliveness of SUT, this approach also checks that the correct responses (state transitions) are observed on the SUT and that the data provided in these responses is valid. Checks on the SUT are performed using basic protocol messages to verify the SUT is alive and in correct state. To describe the SUT behavior in [3], a state-machine is learned from observing the protocol implementation. Messages are associated to different simulated clients and separate state-machines are upheld for each, to check responses and transactions against. Different combinations and mutations of messages are used to provide fuzzed messages where different field values are modified using protocol knowledge for invalid messages and where some field can be repeated or otherwise the overall structure fuzzed.

A method coverage analysis for protocol fuzzing is presented in [4]. Constraints for message formats and processing are defined based on protocol specification analysis. The constraints are formally specified and a test generator is used to generate fuzz tests to cover them. Sometimes effectively fuzzing different parts of the protocol and interactions may require accessing deeper parts of the protocol state-machine, requiring techniques similar to [3] in initiating the protocol to initial phases before fuzzing.

A fuzzing tool for SIP softphones is presented in [14]. This is based on defining templates that identify specific data values to fuzz for different SIP messages. The SUT is then driven to specific states using scripts and these fuzzed messages are injected at these locations in the protocol flow. Additional generic SIP specific fuzzing algorithms are also used, such as re-ordering of SIP headers and defining patterns of specific SIP data vulnerabilities. Finally, the SUT is monitored to evaluate whether it crashes or produces invalid responses.

In [15], performance and robustness testing are combined to evaluate robustness of the system under heavy load. Different types of attacks against SIP based systems are defined, a specific valid load is generated on the system, after which different attack types are launched. System performance is measured before, during, and after the attack. The results indicate system performance in face of attacks under different loads both temporarily and long term.

III. SIP Tester Architecture

To address both the need to test the different configurations (A-D) described in Figure 2, and the different types of testing we are interested in (conformance, performance, robustness) we have to consider both a test framework architecture for executing tests and a set of different test models for generating tests. The architecture needs to support both manual test creation and execution, as well as test generation (and execution) from test models. This means considering the different aspects similar to discussed in [1] but also considering a broader context of testing interacting systems and not just the protocol stack. The following subsections describe our test framework architecture, which is illustrated in Figure 4 (S illustrates the shared state). The test models will be described in Section IV.

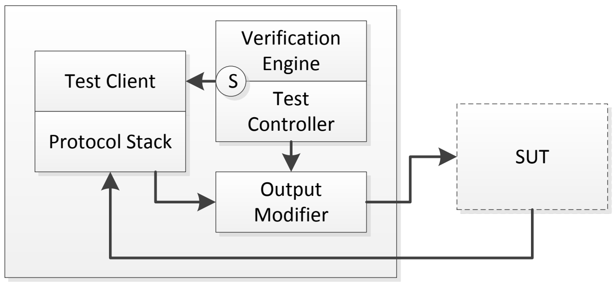


Figure 4. Component Architecture.

A. *Protocol Stack*

To be able to use the tested protocol and evaluate the SUT against it, the test framework also has to implement the basic protocol stack. This consists of parsing and creating messages, and delivering these across the network. Optimally, existing code or an available stack such as an open-source implementation can be re-used. However, sometimes this is not possible due to issues related to the test framework requiring high degree of control and observability over the protocol, which may not have been factors in building some of the available tools and libraries.

In our case, we have implemented our test environment in Java and used the Jain-SIP protocol stack implementation [16]. In this case, our experience has been that using this type of an open-source stack can save some time initially in getting started, but the control and visibility over protocol details is limited. Also, some of the functionality is limited such as lack for proper SIP authentication support. Lack of control and visibility is due to regular user not requiring detailed access to protocol manipulation, and attempting to abstract some parts of the protocol from the user where possible, such as SIP dialog control. However, in order to build a protocol tester, we need to understand the protocol and be able to verify its specifications in sufficient detail. For this reason, as we need to know and understand the details anyway, we find it better to actually implement the stack, at least for the most parts on our own.

As such, we conclude that to have a robust and powerful base for a protocol tester we need our own highly reliable, configurable and observable stack. In an optimal case, we can write one. In practice, resources for this may not be available especially for more complex protocols. In such cases, we need to look at our options with available stacks or with directly using an existing protocol client (such as a SIP softphone or SIP tester such as SIPp [9]).

B. *Test Adapter*

On top of the protocol stack, a test adapter capable of performing stateful transactions against a test target has to be implemented. The test adapter should support higher level functions such as initiating and terminating calls, managing responses from the SUT, and maintaining the protocol state for itself according to these actions and responses. In this sense, it is similar to a SIP softphone but does not have to implement a user interface as it is controlled by the test tool.

There are many existing SIP softphones available (such as Twinkle and Linphone), and these can be used if programmatic control over them is available. Their usefulness depends largely on the extent of remote control supported, and the ability to observe details about the results and responses from the SUT. In our experience, most of the actual SIP clients have limited support or no support at all for such control. However, SIPp is a SIP performance test tool that does provide many such features. While it is limited in its support for detailed control and visibility for conformance or robustness testing, it can be a useful starting point for fast test automation prototyping for suitable parts.

Optimally, the test adapter provides a simple and fast network interface to control it, create protocol messages, and receive notifications about SUT responses. Separating the adapter from the rest of the test framework as a separate networked service allows for using any tools available to implement it and to reuse the controllers and output modifier(s), as well as any existing tools and libraries for different platforms to build adapters. For example, SIPp provides a UDP communication and control interface, and similar interfaces are also used by other successful test frameworks, such as Selenium Webdriver for web applications (which creates and sends JSON requests over the network). This also allows building different controllers on different platforms, using the same adapters, when required.

To summarize, optimally the test adapter provides a stateless control and observation interface to the underlying protocol. This allows the test controller to create various types of protocol messages and observe the results at a selected level of detail.

C. *Test Controller*

To produce actual, executable test cases, a test controller is required. In the case of manually scripted test cases, this executes the given scripts using the test adapter. In the case of using a test generator, this generates the scripts based on a test model and executes them using the test adapter. In a distributed multi-client scenario similar to scenario D in Figure **2**, the controller manages several adapters in parallel.

The controller upholds the test state. When testing scenarios B and C, this means keeping track of the current state of the test client. When testing scenario D, this means tracking all the different test adapters and their connections. It shares this state with the verification engine, which makes assertions about the correct responses received from the SUT based on the controller actions. The adapters can be distributed across the network or on a single machine.

D. *Verification Engine*

A central part of testing is the test oracle, which is a component used to verify that the expected properties hold at the selected points of time in the testing process. For different types of tests, different types of verification engines are required. In conformance testing, the received responses are typically checked after specific actions (such as initiating a call) have been performed. In performance testing, we are interested in measuring the response times to the messages and collecting various metrics on the SUT to assess the impact of test load. In robustness testing, we are interested in observing the state of the target system and using this information to make assertions about how invalid inputs impact the SUT state and responses.

The verification engine performs these various checks to evaluate test results during system operation. The checks performed can be split into passive and active checks. Active checks are performed as specific checks at specific points in the test execution, e.g., to establish that response messages such as TRYING, ACK, and OK are received when required and contain the correct data related to their associated requests. Examples of passive checks would be to track that messages that require a dialog should not be received outside dialogs, or to continuously ping the SUT to ensure it is alive.

For active verification, the verification engine has to share state with the test controller to be able to make the required assertions. For some forms of passive verification such as pinging the SUT this is not required but the verification engine still has to communicate back to the test controller to notify of any failed checks. This then fails the executed test and reports the results back to the user.

E. *Output Modifier*

The output modifier is used by the test controller during robustness testing to invoke specific modifications on the input messages produced by the test client. Test data is passed through the output modifier and forwarded to the SUT. During robustness testing, the test controller can enable different types of fuzzing patterns to be applied to the data, while during other types of testing the output can be forwarded as is.

IV. TEST MODELS

This section discusses different types of test models we use for testing conformance, performance and robustness and how these relate to the architecture. The basis is the conformance test model, and the other models specialize and extend it in different ways. These models are also different depending on the type of scenario addressed. For scenario A, the models target the protocol stack functionality, performance and robustness. For scenarios B and C, the models target the interactions of a single client with other nodes in the network. For scenario D, the models target the overall system behaviour. In our testing, we have focused at the level of scenarios B-D, and assume that the stack will be tested sufficiently as part of these test cases and separately at unit and component testing level by the developers.

While describing these as test models implies our preference towards model-based test generation, it is equally possible to use the information in the models to create test cases manually. The test model is used by the test controller as a part of the overall test framework.

A. *Conformance Test Models*

The conformance model describes the expected functional behavior of the SUT as described by its specification. In the case of scenarios B and C, this is the SIP RFC 3261 [17]. In scenario D, the specification describes the expected behaviour of the overall system and its interactions. Table I lists the basic rules and actions for the scenario B and C model. Table II lists the basic rules and actions for the scenario D model.

TABLE I. SCENARIO B AND C MODEL ELEMENTS.

ID	**Rules**	**Action**
S_R	Unregistered	Register
S_U	Registered	Unregister
S_I	Registered, No Call	Invite
S_C	Registered, Calling	Cancel
S_O	Registered	Options
S_B	Registered, Call On	Bye
S_M	Registered	Message

The actions in Table I are basically the SIP request messages as described in [17]. For scenario B and C, the test model is focused on generating requests to interact with the SUT. In addition to this, the test framework must handle the responses from the SUT, such as failures, errors, and successes. It must also provide its own responses to such messages when required, such as the ACK message to the OK for INVITE. In our test framework architecture, the test controller executes the tests and maintains the relevant state information to manage the responses. The state information describes the protocol interaction state including registration status, call invite status, and ongoing call status.

Additionally, the model has to define the valid data values in order to properly evaluate the conformance of the system and to expect the correct responses.

TABLE II. SCENARIO D MODEL ELEMENTS.

ID	Rules	Action
SD_R	Phones < MAX_P	Register Phone
SD_U	Phones > 0	Unregister Phone
SD_I	Free Phones >= 2	Initiate Call
SD_T	Busy Phones > 0	Terminate Call
SD_S	Servers < MAX_S	Start Server
SD_X	Servers > 0	Stop Server

For scenario D, our model is focused on testing high-level interactions of different communicating entities in the overall system. As illustrated in Figure **2**-D, there may be several servers and phones or other SIP user agents in the system, connected in various ways. The actions shown in Table II allow dynamic creation of such configurations and to establish and terminate connections between the nodes. Test framework/model state in this case consists of the nodes and their status. The state for SIP clients is the same as for scenario B and C, including the connected nodes in a call. The SUT in this case is the overall system interactions. The model of a SIP phone described in Table I can be used to represent a phone, which is controlled by an overall system test model.

The test oracles for the verification engine in the conformance model are checks of the test model state against the SUT state. This means we will check that when the SUT should accept a call, the *invite* message passes and the call is established. Similarly, when registration should succeed, the response is expected to be a success. After an *invite* has been performed but before the call is started, a *cancel* message should stop the call from starting. Once a call is started, a *bye* message should stop the call and allow re-starting a new call with another *invite*. Similar checks are performed at every point during the execution of a generated test case with regards to every request message performed, and every response message received. As the test controller maintains the system state according to performed actions, it can automatically verify all these properties with minimal effort.

B. *Robustness Test Models*

We define robustness according to the IEEE glossary as "The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions." [18]. In our robustness models, we consider both invalid interaction sequences as well as invalid input data. We represent the interaction sequences with similar models to the conformance model, and the input data using the output modifier configured with message modification patterns.

Defining invalid inputs and their expected responses explicitly can be challenging as many protocol definitions have required parts (must) and optional parts (may). In many cases, also ambiguities exist and different implementations have taken different interpretations of these. Due to this, we classify any issues observed either as warnings or errors. Warnings are issues related to potential issues, while errors indicate clear problems in the implementation.

Our robustness models specialize the conformance models by changing specific parts of the models to produce invalid interaction sequences and data. We call these specializations *robustness model patterns*. To do this, several mechanisms of our MBT tool [19] are used. These are *model composition*, *startup-sequences*, and *model slicing*.

Model composition refers to combining several separate model objects together. The normative model elements are expressed using the conformance models as shown in Table I and Table II. The invalid sequences are in a separate model object that is combined with the conformance model by the test generator. If we call the conformance model C and the robustness model R, the actual base test model T is their union, $T = C \cup R$. Example robustness rules and actions for the model in Table I are shown in Table III.

TABLE III. ROBUSTNESS MODEL ELEMENTS.

ID	Rules	Action
SR_R	Registered	Register
SR_U	Unregistered	Unregister
SR_I	SUT in call	Invite
SR_C	SUT not called	Cancel
SR_O	Not registered	Options
SR_B	No call with SUT	Bye
SR_M	Not registered	Message

Startup-sequences are used to initiate the SUT into a state of interest for the robustness test pattern. These can be used to drive the initial test generation to a specific state by making the generator take a specific set of steps before starting its own algorithmic generation. For example, we can define one from the model in Table II as "Register, Invite", which means the generator will start all test cases with this sequence and thus the tests will start from a valid registered state with a valid initiated call started with the SUT.

Model slicing allows us to define which parts of the model are to be used for generation and how much they are used. If we take the test model $T = C \cup R$, the sliced model S is then a subset of T, $S \subset T$. The slice can either remove a step from T completely or limit the number of times it can appear in T. The slice does not affect the startup sequence and the startup sequence does not affect the slice, allowing these to independently define different elements of the robustness test scenario. For example, the slice may forbid any *invite* messages but the startup sequence can still use them as the slice only affects parts after startup.

As an example, let us show a pattern for robustness testing registration handling for a single node during an ongoing call. This is illustrated in Table IV. In this case our model is $T = C \cup R$ as discussed, where C equals the model shown in Table I and R equals the model shown in Table III. Using this pattern configuration, the generator will generate sequences that always start with valid *register* and *invite* messages. This is then followed by any allowed messages except *bye*, which is forbidden by the slice. Notice that this pattern also includes and allows all steps in R.

TABLE IV. EXAMPLE ROBUSTNESS PATTERN.

Element	Value
Startup	S_R, S_I
Slice	!S_B

Additionally, the output modifier patterns change the created messages in various ways:

- Duplicate headers and message parameters
- Remove headers and message parameters
- Modify headers and message parameters

When running robustness tests, the test oracle definitions require some special attention. We can define how each of the invalid input producing steps should impact the SUT operation and state. Typically this would be to ignore the input with invalid data or sequences. In other cases we can also define specific impacts and update state accordingly. For example, if we consider security vulnerability scanning as part of robustness, some specially crafted input for such tests can be considered valid but should have no unwanted side-effects. In these cases, we should update the state in those steps, and evaluate the oracles accordingly.

However, due to different specification interpretations or desire to provide flexibility in communicating with other endpoints of varying quality, the responses to some of the robustness input may differ, and the SUT may accept some of them as valid. For example, duplicate headers produced by the output modifier may be interpreted as an issue or not in the SUT. In such cases, we can choose to disable some of the more strict oracles for those models and tests generated from them and focus on the more generic ones to check the system for generic properties such as not crashing or ending up in a bad state for any node, or consume excess resources over time.

For such cases, the test oracles that make such assertions can also be represented as their own model object(s). The test model T then becomes $T = C \cup R \cup O$, where O is the model object holding these oracles. By removing O, or parts of it, from the equation, the oracles can be disabled as required. In any case, as mentioned the generic test oracles are always valid, such as pinging the SUT and checking error codes. These can also be configured to run at specific intervals to check for general properties in, e.g., long running performance tests.

C. *Performance Test Models*

Our performance test models are combinations of different configurations of the conformance and robustness test models. They are intended to explore the performance limits of the SUT under different environment and load conditions. They represent different usage scenarios for different types of user profiles in the system. Basically we use our conformance test models as valid client type and the robustness model instances as another (invalid) client types.

Similar to [7], we use different types of traffic patterns for specific user profiles, such as traffic bursts and linear increase in traffic. We start with our conformance test model clients as the reference set of providing the system performance under these different types of varying load. Once we have this model, we apply our different types of robustness test patterns as clients to represent invalid data, similar to attacks discussed in [15]. Finally, we re-run the initial reference test set with the conformance test clients for valid data and compare the results with the initial run before invalid data was used. We then use these results to give us a model for the overall system performance under different types of load.

V. DISCUSSION

While we have described a composition of model objects as one for the conformance test model and another for the robustness model, and using model composition and scenario slicing to create robustness patters, it is possible to further decompose these models as much as desired. The actual composition we support is not limited in the number of model objects and thus the operation can be seen as $T = C_N \cup R_N \cup O_N$, where N refers to having any number of these in the end result. However, in practice we have found that a smaller number of model objects makes it much easier to manage the overall set of patterns. For a protocol such as SIP, where there is a relatively small set of potential messages having one C, R, and O has worked well for us. For more complex protocols it may be necessary to split these further, for example, to make model composition and slicing for different test purposes and patterns easier.

As it is, in the work presented in this paper we have so far focused on the SIP protocol. More generally, we see the approach applicable more widely to different protocols and networked systems. The architecture, conformance and robustness models, and robustness pattern definitions simply need to be adapted to the new specifications. This means creating suitable protocol adapters, and defining the valid and invalid sequences to be used for test generation. That is, the overall framework and modelling approach is intended to be easily specialized for a variety of protocols.

So far our test execution and generation has focused on a single host environment. For now we have found this to be sufficient for our testing needs, as modern systems can run numerous clients in parallel even on a single multi-core system. However, more distributed systems are needed to address more realistic usage scenarios as well as to scale to very large scale testing. This would also include modelling different concurrent users more realistically in terms of latencies, burst traffic, occasional robustness scenarios interleaved with conformance scenarios, and other similar attributes. In our previous work, we have investigated distributed model-based test generation [20]. In the future we hope to extend also our test framework to make use of this type of strategies, including distributed (cloud) deployments.

Another interesting point of extension for this work is to include actual specific security related attacks to the robustness patterns. Currently we mainly use fuzzing related patterns, which change the inputs in different ways and evaluate the SUT robustness. Additionally, specific inputs to target specific vulnerabilities in underlying backend systems could be of interest. Generally, we are also looking at extending our work to cover more aspects as well, such as quality of service for the call under different conditions.

When executing large scale test cases, the biggest issue we observe is making reliable overall assertions about the system state. For this reason we have at large scale mostly focused on observing overall performance across large sets of users. However, when issues are observed from such large scale tests, debugging them for root cause analysis can be very challenging due to large numbers of different types of interacting clients and servers. While these issues are not specific to our approach but to large scale testing in general, in the future we hope to explore better solutions to these issues as well and integrate these into our approach as easily applicable solutions.

VI. Conclusions

In this paper, we have presented an architecture, a set of test models and ways to compose and slice these to form a holistic test framework for the SIP protocol. Our framework supports conformance testing, robustness testing, and performance testing, with each part building on top of the previous, allowing for an effective and extensive implementation. We are currently extending the work by collecting a wider set of patterns building on top of the framework presented in this paper, as well as applying these in industry case studies. In the future, we are interested in refining this work based on practical applications and experiences, and extending it to more diverse set of protocols.

VII. References

[1] W. Grieskamp, N. Kicillof, K. Stobie and V. Braberman, "Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology," *Journal of Software Testing, Verification and Reliability,* vol. 21, no. 1, pp. 55-71, 2011.

[2] M. Sutton, A. Greene and P. Amini, Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley, 2007.

[3] H. J. Abdelnur, R. State and O. Festor, "KiF: A Stateful SIP Fuzzer," in *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications (IPTComm2007),* 2007.

[4] P. Tsankov, M. T. Dashti and D. Bashin, "Semi-Valid Input Coverage for Fuzz Testing," in *International Symposium on Software Testing and Analysis (ISSTA2013)*, 2013.

[5] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufman, 2006.

[6] C. Caba and J. Soler, "An IMS Testbed for SIP Applications," in *Principles, Systems and Applications on IP Telecommunications - IPTComm '13*, 2013.

[7] L. Roly and L. Schumacher, "SIP Overload Control Testbed: Design, Building and Validation Tests," in *IEEE Consumer Communications and Networking Conference,* 2011.

[8] SailFin, "SailFin Project," [Online]. Available: https://sailfin.java.net/. [Accessed 23 4 2014].

[9] SIPp, "SIPp," [Online]. Available: http://sipp.sourceforge.net. [Accessed 23 4 2014].

[10] F. Lalanne and S. Maag, "A Formal Data-Centric Approach for Passive Testing of Communication Protocols," *IEEE/ACM Transactions on Networking,* vol. 21, no. 3, pp. 788-801, 2013.

[11] D. Bao, D. C. Carnì, L. D. Vito and L. Tomaciello, "Session Initiation Protocol Automatic Debugger," *IEEE Transactions on Instrumentation and Measurement,* vol. 58, no. 6, pp. 1869-1877, 2009.

[12] P. Xiong, C. Pu, X. Zhu and R. Griffith, "vPerfGuard : an Automated Model-Driven Framework for Application Performance Diagnosis in Consolidated Cloud Environments," in *International conference on performance engineering (ICPE '13)*, 2013.

[13] M. H. Sørensen, "Use Case-Driven Performance Engineering without "Concurrent Users"," in *International Conference on Performance Engineering (ICPE '13)*, 2013.

[14] S. Taber, C. Schanes, C. Hlauschek, F. Fankhauser and T. Grechenig, "Automated Security Test Approach for SIP-Based VoIP Softphones," in *Internation Conference on Advances in System Testing and Validation Lifecycle (VALID2010)*, 2010.

[15] P. Steinbacher, F. Fankhauser, C. Schanes and T. Grechenig, "Work in Progress : Black-Box Approach for Testing Quality of Service in Case of Security Incidents on the Example of a SIP-based VoIP Service," in *Principles, Systems and Applications of IP Telecommunications (IPTComm2010),* 2010.

[16] "JSIP: Java API for SIP Signaling," [Online]. Available: https://jsip.java.net/. [Accessed 24 4 2014].

[17] IETF, "RFC 3261 SIP: Session Initiation Protocol," IETF, 2005.

[18] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990,* 1990.

[19] T. Kanstrén, "OSMO Tester Home Page," April 2014. [Online]. Available: http://code.google.com/p/osmo. [Accessed April 2014].

[20] T. Kanstren and T. Kekkonen, "Distributed Online Test Generation for Model-Based Testing," in *Asia Pacific Software Engineering Conference,* 2013.

Performance Impact of Correctable Errors on High Speed Buses

Daniel Ballegeer, David Blankenbeckler, Subhasish Chakrborty, Tal Israeli
Intel Corporation
Santa Clara, CA, USA
Emails: {dan.g.ballegeer, david.blankenbeckler, subhasish.chakraborty, tal.israeli}@intel.com

Abstract— **Modern high speed serial buses are generally required by specification to achieve a maximum bit error ratio. Are these requirements too restrictive? This paper will look at a series of studies on Peripheral Component Interconnect Express and Serial AT Attachment, investigating the impact of bit error ratio on bus performance. The results of these studies suggest that typical bit error ratio requirements may be conservative. The findings suggest that alternative bus performance specifications should be considered that would open new possibilities for design, validation and manufacturing test tradeoffs.**

Keywords-bit error ratio; BER; electrical validation; high speed interconnect; high speed bus; I/O.

I. INTRODUCTION

Modern high speed serial bus specifications generally have a requirement for maximum Bit Error Ratio (BER) [1][2][3][4]. In this context, bit error ratio is defined as the fraction of bits transmitted over the high speed interconnect that are interpreted incorrectly at the receiving device—i.e., a bit originally transmitted as a "1" is interpreted as a "0" or vice-versa. Table I summarizes these for a variety of buses: Third Generation Peripheral Component Interconnect Express (PCIe Gen 3), 10 Gigabit Ethernet, Serial AT Attachment (SATA), and Universal Serial Bus (USB). Note that there is no inherent need or expectation that each interface type has the same BER requirement, but the table illustrates that 10^{-12} is quite commonly used.

Many high speed buses such as the ones listed in Table I utilize error detection schemes such as a Cyclical Redundancy Check (CRC) at the receiving device to detect any signal integrity-induced bit errors that could have occurred over the interconnect. In such a scheme, in the event of a detected error, a request is sent to the transmitting device to send the data again (a retry). Ideally, a target BER level on an interconnect that employs a CRC check must take into account both the effectiveness of the CRC scheme with respect to the protected data packet size as well as the performance losses that result from error-induced retries on the bus. Although studies and publications on the effectiveness of CRC error detection have occurred for multiple decades [5][6], as far as the authors know, there have been few, if any, studies done on real world performance impact at various error rates. Some theoretical calculations of latency impact vs. error percentage have been presented [7], but this would not take into account other factors that interact with the error retries and contribute to the overall performance impact on a true workload. This paper will outline the results of several studies conducted to better understand the real world performance impact of increasing error rates beyond the specification level.

TABLE I. BER SPECIFICATIONS FOR SOME HIGH SPEED BUSES

Link	BER Spec
PCI Express Gen 3	10^{-12} [1]
10 Gigabit Ethernet	10^{-12} [2]
SATA 3.x	10^{-12} [3]
USB 3.x	10^{-12} [4]

These studies are interesting in that they provide some data justifying room for design tradeoffs. For example, there may be significant cost savings opportunities, trading off slight performance impact for lower cost material. Consider the example of a system design with long Peripheral Component Interconnect Express (PCIe) bus routing lengths. Instead of using more expensive low loss Printed Circuit Board (PCB) material, it may make sense to sacrifice error rate and realize a cost savings with standard FR4 PCBs. Likewise, there may be power reduction opportunities for low power devices, trading off performance for lower power operation, without sacrificing data integrity.

It is easy to show through either empirical measurements or theoretical arguments that the bus BER of a product is a distribution when measured across multiple instances of that product. Factors that induce this distribution include, among other things, the variations in the characteristics of the board interconnects, receiver circuitry, and transmitter circuitry. For example, in the voltage domain of the bus signal, these factors lead to a distribution of the electrical margin, V_m, where V_m represents the amount of voltage swing at the receiving device beyond the minimum required voltage detection threshold.

To simplify the example, consider an ideal case where there is no noise in the system when V_m is measured, i.e., V_m is the noise-free voltage signal margin. In a real system, bit errors result from noise adding or subtracting from this margin. In a zero-mean additive white Gaussian noise model such as that described in [8], the V_m distribution may be mapped into a BER distribution via the following relationship:

$$BER = \frac{1}{\sigma\sqrt{2\pi}} \int_{V_m}^{\infty} e^{-\frac{x^2}{2\sigma^2}} dx \,, \qquad (1)$$

where σ is the standard deviation of the Gaussian noise.

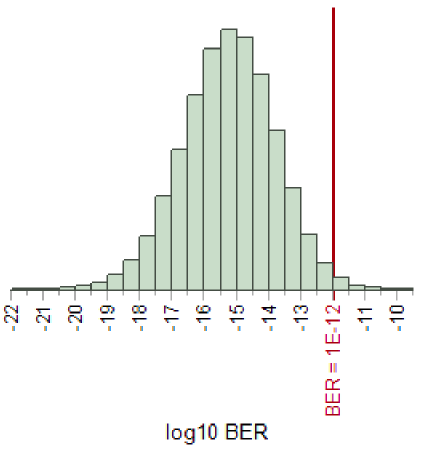


Figure 1. Conceptual illustration of bus BER distribution across different units

Fig. 1 is an example illustration of a BER distribution that could result from a Gaussian distribution of V_m. The specifications are a hard cut off at maximum BER limit, like 10^{-12}. In reality, the BER performance of every lane on every channel on every board is different. Is it acceptable that a small portion of lanes are slightly above the BER spec if the performance impact is negligible? Are these systems really considered bad if there is no noticeable performance impact to the end user? If it were acceptable to ship some portion of systems at a higher BER, there may be substantial benefit, such as the opportunity to reduce silicon test time requirements.

II. Performance Impact Experiment Overview

In order to measure the performance impact of BER levels above specification, four different high speed serial bus usage scenarios were studied:

- a PCIe Gen 3 bus used with a graphics add-in card
- a PCIe Gen 3 test add-in card utilized for easy measurement of data mismatch errors
- PCIe Gen 3 used as an interconnect between a CPU and Platform Control Hub (PCH)
- a Serial ATA (SATA) 6 Gb/s interconnect attached to a hard disk drive.

In all experiments, techniques were used to induce different BER levels on the link, either by

- changing the voltage or timing sampling at the receiving device to be offset with respect to the data eye center
- error injection at the receiver, or
- voltage swing attenuation at the transmitting device.

Then, with this induced BER present, performance benchmarks were run that specifically focused on the I/O being studied. In some cases, the BER was able to be monitored at the same time as the performance, whereas in other cases, BER had to be measured first in a loopback scheme before running the performance benchmark at the same settings. In all cases, experiments were re-run at least one time to confirm the performance results quoted.

III. Performance Data Collection and Results

A. *PCIe Gen 3 used with a graphics add-in card*

In the first experiment, the PCIe Gen 3 bus studied was the interconnect between a 3rd Generation Intel Core i7 Processor and a PCIe graphics Add-In Card (AIC). Four different high-performance graphics cards were included in the experiment, spanning three different vendors. Graphics card settings were set to produce maximum performance; future studies will also include studies with the scenario where hardware acceleration is turned off. Three different commercially available graphics-intensive benchmarks were run during the experiment: Codemasters Dirt 3, a graphics-intensive racing game; Unigine Heaven, a graphics-intensive benchmark designed to stress graphics AICs, and 3DMark Fire Strike, a real-time graphics rendering benchmark. In this experiment, the degradation of performance vs. BER was measured in both directions: in one set-up, the CPU receiver experienced the bit errors, and in a separate set of measurements, the graphics AIC receiver experienced the bit errors.

The CPU PCIe Receiver (Rx) circuitry had built-in validation test hooks that allowed changing the location of the sampling point in the time domain with respect to the data eye center. By offsetting the sampling point away from the data eye center, bit errors could be induced at the receiver.

In the first step, the BER vs. time sampling offset was established by sending a known random bit sequence out the CPU transmitter and receiving the same bit sequence at the CPU receiver. This was accomplished via the far-end digital loopback mode supported by all PCIe spec-compliant components. In this mode, the AIC received and interpreted the data transmitted by the CPU and then retransmitted the same data back to the CPU. Received data at the CPU was compared to the CPU transmitted data to detect the level of bit errors at each sampling offset point. Note as sampling offset moved closer to nominal data eye center, more transmitted bits were necessary to detect bit errors. BER vs. sampling offset slope was checked to ensure the relationship agreed with an additive white Gaussian noise model indicative of Random Jitter (RJ).

In the second step, the identical set of time sampling offset values were used in the same system setup, this time allowing the three benchmarks to run and measure performance. In this way, a correlation of performance vs. BER at the CPU receiver could be established.

To establish the relationship between performance impact of bit errors received by the Graphics PCIe Rx, a slightly different approach was used to induce bit errors: the CPU transmit voltage swing was reduced incrementally to produce different levels of bit errors experienced at the receiver of the graphics card. The relationship of BER level vs. transmit swing was first established by a loopback testing mode on the system similar to the one described above.

Then, using the same transmit swing settings, the performance of each of the three benchmarks was measured. Using this approach, performance vs. BER experienced at the graphics card receiver could be characterized.

TABLE II. MINIMUM BER LEVELS AT GRAPHICS RX THAT INDUCED 3% AND 50% PERFORMANCE LOSS ON WORST-CASE BENCHMARK

Graphics Card	BER for Graphics Rx performance loss	
	3%	*50%*
A	$1x10^{-8}$	$1x10^{-6}$
B	$3x10^{-6}$	$6x10^{-5}$
C	N/A	N/A
D	N/A	N/A

Table II summarizes the performance loss observed as a function of BER at the Graphics Rx. Table III lists similar information as a function of BER at the CPU Rx. Values are reported for both 3% performance loss and 50% performance loss. Note that cards C and D were extremely robust to low CPU Tx voltage swing and did not encounter bit errors even at the lowest swing settings. Therefore it was impossible to characterize performance vs. Graphics AIC Rx BER on cards C and D using this technique.

TABLE III. MINIMUM BER LEVELS AT CPU RX THAT INDUCED 3% AND 50% PERFORMANCE LOSS ON WORST-CASE BENCHMARK

Graphics Card	BER for CPU Rx performance loss	
	3%	*50%*
A	$1x10^{-7}$	$3x10^{-6}$
B	$1x10^{-4}$	$2x10^{-4}$
C	$5x10^{-7}$	$2x10^{-6}$
D	$4x10^{-8}$	$3x10^{-7}$

The first notable point is that even a 3% performance loss was not observed until a BER of at least 10^{-8}, which is four orders of magnitude above the PCIe BER spec of 10^{-12}.

Second, although the table values represent the worst-case benchmark, there was not a large difference in the behavior of different benchmarks in terms of relative performance loss. This is shown in Fig. 2, which depicts the BER vs. performance loss at the CPU receiver when using PCIe card D. Also evident in Fig. 2 is the typical number of BER sample points and intervals that produced the data summarized in Tables II and III. This card showed the most difference between benchmarks, but as can be seen, even on this card, the relative performance loss is roughly equivalent across all three benchmarks at a given BER.

Another observation is that once performance starts to degrade on the order of 3%, it does not require a much greater BER to degrade the performance significantly further. This can be seen in Tables II and III, or graphically in Fig. 2. The latter graph illustrates that each benchmark performance metric degrades by 50% at a BER only 1 to 1.5 orders of magnitude above the 3% degradation point.

However, there were some differences in CPU Rx BER and Graphics Rx BER in this regard. Fig. 3 depicts this finding for Card A running Unigine Heaven. Graphics Rx BER starts to produce performance problems at a level roughly two orders of magnitude below CPU Rx BER, but the performance decrease after that point is more gradual than CPU Rx, such that at BER levels in the vicinity of 10^{-6}, performance penalties are similar.

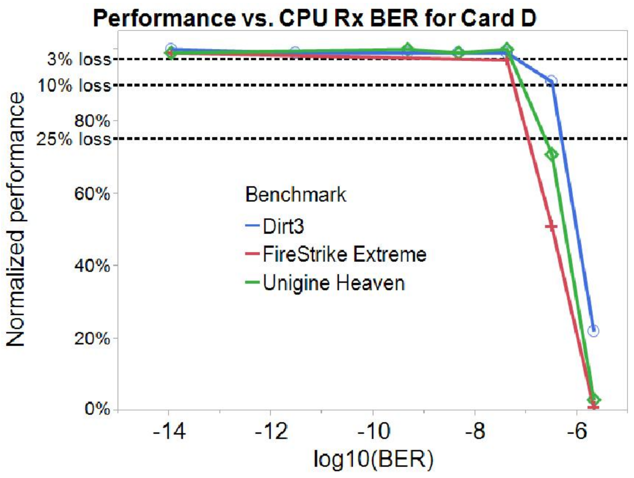


Figure 2. PCIe Card D performance vs. BER on each of three benchmarks

It should also be mentioned that some card-to-card differences were observed. This is shown in Fig. 4, which separately delineates the Unigine performance vs. CPU Rx BER for each card. Although card B had the lowest performance, it proved to be the least affected by bit errors, with little degradation all the way up to 10^{-4} BER. It could be speculated that the lower performance of this card resulted in lower utilization of the maximum available bandwidth on the PCIe link, thus preserving some additional bandwidth to compensate for the error retries on the link. However, this would not explain why card A, the highest performing card, showed the second-most resilience to bit errors in terms of performance impact. This suggests there are other factors that create these differences from card to card.

B. PCIE Gen 3 link between CPU and test add-in card

Another round of experiments was designed with a PCIe Gen 3 test add-in card to understand the BER levels associated with serious performance degradation. Voltage sampling and timing sampling points on the CPU PCIe receiver were offset from nominal values to induce a bit error ratio in the digital loopback mode described in the previous section, in order to establish the relationship of BER to the margin offsets.

Next, a PCIe functional test mode was utilized, in which the CPU wrote pre-defined data to the add-in card with all sampling points at nominally trained values. While reading back the data from the card, the CPU receiver margin hooks were operating to test at different sampling offset points, and error reporting was enabled to give visibility into detected receiver errors such as bad packets and CRC errors. In addition, data mismatch errors escaping the PCIe error detection mechanisms were identified by comparing the received bits against the transmitted bits in the CPU memory. In this way, the BER level creating normally undetected data mismatch errors could be empirically measured. The experiment was performed once with the timing sampling offset used to induce BER, and again with the voltage sampling offset used to induce BER at the CPU receiver.

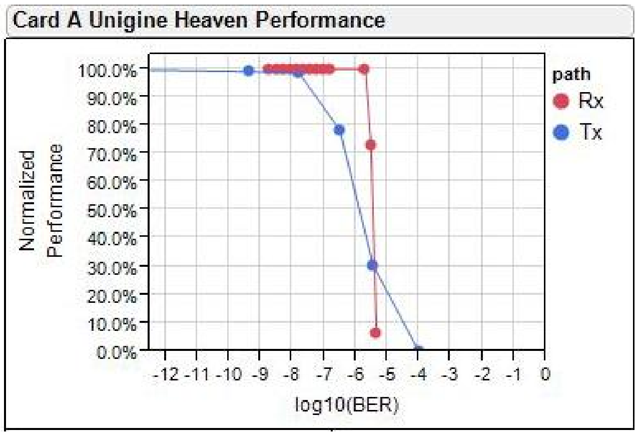


Figure 3. Unigine performance vs. BER on either CPU Rx or Tx vs. BER on PCIe graphics Card A

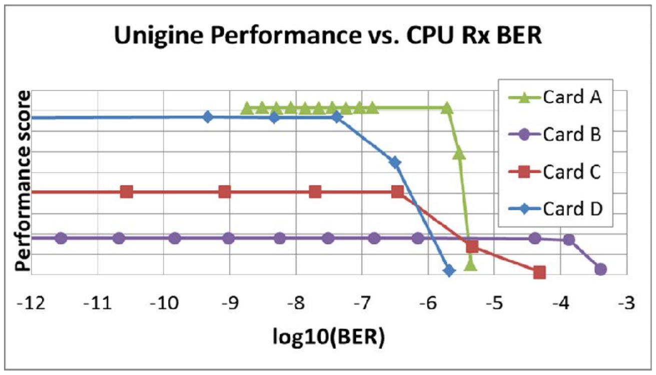


Figure 4. Unigine performance vs. CPU Rx BER on each of four PCIe graphics cards

Table IV shows the comparison of BER levels resulting in data mismatch errors during the PCIe functional test versus the BER levels causing a 100% performance loss. It should be mentioned that with the PCIe functional test content running in this part of the experiment, the 100% performance loss in actuality resulted in a crash or link hang requiring a reboot.

When BER was induced by changing the timing sampling point, the resolution was not sufficient to distinguish any data mismatch errors before reaching a BER that caused 100% performance loss. When using the voltage sampling offset, on 8 of the 120 runs, data mismatch errors were distinguishable before a crash occurred. On the other 112 runs, the high level of BER created a crash before any mismatch problems occurred.

Evident from these results is that any data mismatch issues escaping the built-in error detection mechanisms on PCIe Gen 3 occur at a BER very close to or higher than the BER that causes catastrophic performance problems. This is supporting evidence that as BER is increased, the main area of concern for an end user is in fact performance degradation rather than undetected data mismatch issues.

C. PCIe Gen 3 link between CPU and PCH

In this experiment, a 2nd generation Intel Xeon E5 processor was connected to an Intel BD82C606 Server Chipset Platform Control Hub (PCH) via a PCIe Gen 3 uplink. The intent was to study the impact of PCH Rx BER on performance of the uplink.

TABLE IV. AVERAGE BER AT WHICH 100% PERFORMANCE LOSS OR DATA MISMATCH ERRORS OCCURRED ON PCIE GEN3 ADD-IN CARD

Method used to induce BER	Avg BER for 100% performance loss	Avg BER for data mismatch error
Timing sampling offset	3.0×10^{-7} (120 runs)	Not measurable
Voltage sampling offset	4.8×10^{-7} (120 runs)	5.6×10^{-7} (measurable on 8 of 120 runs)

First, in order to monitor the performance, a benchmark test was run that was known to exercise the bandwidth of the PCIe link. While this was done, jitter of various amplitudes was injected at the receiver to induce a BER at the PCH Rx. While the jitter was injected, error logs were utilized to monitor the rate of CRC and link recovery errors with respect to the total number of bits transmitted to calculate the effective BER at that jitter amplitude setting. It was found that the jitter injection provided only a coarse control over the effective BER. Finer granularity was achieved by complementing the jitter injection with voltage and temperature adjustments, which provided a finer adjustment to the receiver BER level. This way, performance penalty vs. PCH PCIe uplink receiver BER could be characterized.

The jitter injection technique plus voltage & temperature adjustment did not provide as fine of control over the BER as the sampling point adjustment technique used in part A. However, this technique did have the advantage of being able to monitor the actual bit errors occurring during the performance test runs themselves.

Fig. 5 displays the results of this experiment. The region of >3% performance penalty was witnessed to be in the vicinity of 10^{-10} BER, again implying there is some buffer between a performance issue and the 10^{-12} BER specification. Because of lack of precise control over the BER with the jitter injection, there was a clear absence of data points in the BER range of 10^{-9} and 10^{-4}. Somewhere in this range, and by the time 3×10^{-4} BER is reached, the part is not able to function, which is represented by the 100% performance penalty on the graph in Fig. 5. Because of the sparseness of the data points, it is not known at exactly what BER this occurs. Based on the slope of the points at or around ~10^{-10}, it appears that 50% degradation would occur in the low 10^{-9} range. This agrees with the PCIe graphics card experiment in section A, which also showed a performance degradation from 3% to 50% occurring within approximately 1-1.5 orders of magnitude change in Rx BER.

D. SATA 6Gb/s link between PCH and hard disk drive

For this measurement, an Intel BD82C606 Server PCH SATA 6 Gb/s link attached to a hard drive was studied. Similar to the experiment in the previous section, jitter injection was used at the PCH Rx to induce a BER. Jitter frequency and amplitude changes were made to vary the BER, and for finer adjustments, temperature adjustments were made in addition to a validation test hook that provided some level of control of the PCH Rx voltage sampling point with respect to the center of the data eye. As these adjustments were being made, the BER could be calculated by logging the disparity and CRC errors occurring on the SATA link and dividing by the total number of bits transmitted.

While errors were being induced in this manner, a performance benchmark involving continuous reads and writes to the hard drive was utilized to stress the SATA I/O as well as monitor the performance at various levels of BER. With the combination of jitter injection and the data eye margining hook, a reasonable level of accuracy was achieved in inducing different levels of BER on the SATA link. Similar to the PCH PCIe uplink experiment, errors were induced and monitored while the performance monitor itself was being run.

Fig. 6 shows the outcome of the experimental measurements. As BER was increased above the spec of 10^{-12}, minimal overall performance degradation was witnessed until a BER level of approximately $3x10^{-10}$ was achieved. At this level, a 3% performance penalty was observed, but from that point on, the rate of performance degradation with respect to BER increased dramatically. As was so often witnessed in the experiments reported in this whitepaper, 50% performance degradation occurred only at a BER level one order of magnitude higher, at approximately $3x10^{-9}$.

IV. IMPLICATIONS AND FUTURE WORK

The data presented here suggests for the high speed serial bus types studied, there are at least two orders of magnitude of margin above the max BER specification before a user would experience any noticable performance loss from replaying data after an error is detected. It is worth mentioning, however, that the empirical data sometimes showed lower margin than a simple latency-based theoretical projection would predict. Murali et al. [7] speculate that based on error retry-related latency penalties, average observed latency would not show degradation until a packet or flit error ratio in the range of 0.1%-1%. In this context, latency refers to the amount of additional delay in the data packet, or "flit," that is created by the receiving device notifying the transmitting device of the CRC error as well as the resend of the correct data by the transmitting device. Projecting this value onto PCIe Gen 3, for example, with a typical CRC-protected packet payload size of 1200-2200 bits for the products measured in this paper, one would predict there would be no performance concern until a BER elevates to the range of $\sim 10^{-7}$ to 10^{-6}. Yet in some of the experiments, performance began to show measurable decrease in the neighborhood of 10^{-8} or even 10^{-10}.

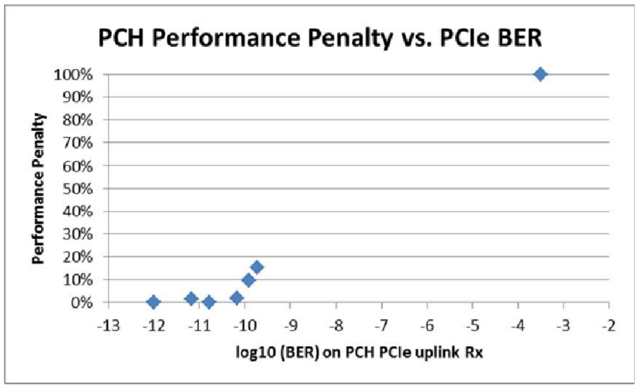


Figure 5. PCH performance penalty vs. PCH Rx BER on the PCIe uplink to the CPU

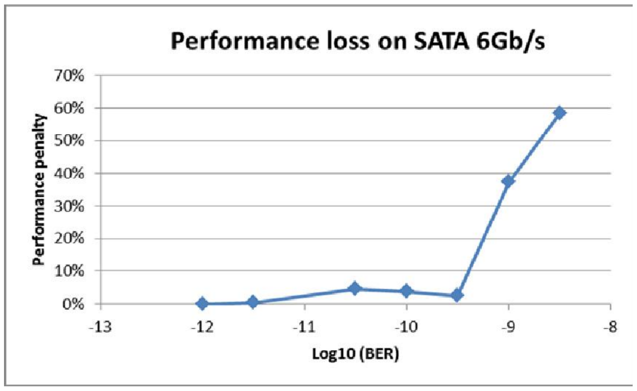


Figure 6. Performance loss vs. PCH Rx BER on the SATA 6Gb/s link to a hard drive

This suggests that true effective latencies with real modern-day products and workloads, taking into account the error profiles (for example, number of consecutive packets with errors), are sometimes greater than the assumed penalties in [7]. While error ratio profiles could differ by scenario and would not always match those in the reported experiments, the fundamental sources of bit errors in the experiments (jitter, elevated temperature, reduced transmit voltage swing, and a non-centered data sampling point) are all sources that could be experienced in a real-world system.

To minimize the impact of extended test runs and using more expensive design solutions to ensure parts meet the BER spec, an alternative approach to a simple spec value would be to architect in the right validation hooks and capabilities to measure performance changes as data eye margins decrease or alternatively, as BER increases. Validation activities can then concentrate on checking that the vast majority of parts and systems will not experience noticeable performance penalties—for example, no more than 3% performance loss—from resending data across the link as a result of error detection. When needed, test content such as the PCIe Gen 3 functional test used in this paper can be used to confirm that undetected data mismatch errors happen at or above BER levels that create severe performance degradation or hangs.

By structuring validation targets with respect to performance, product validation teams can have confidence they are truly validating for a quality end user experience, rather than a generic BER level. To illustrate, the BER requirement of 10^{-12} is prevalent in specs for high speed serial buses, despite different levels of error detection and different retry time penalties on these various buses, not to mention product-level architectural differences that could create different retry penalties product to product on the same serial bus type. By forcing products to abide to one generic BER spec that is not explicitly tied to an end user impact, the spec level must be overly conservative to account for all possible factors across all possible systems, implying that most products are over-designing and over-validating.

TABLE V. TEST TIME DIFFERENCES AT DIFFERENT BER LEVELS

BER requirement	Min test time for 95% confidence, PCIe G3 (seconds)	Min test time for 95% confidence, SATA 6 Gb/s (seconds)
10^{-12}	374	499
10^{-10}	3.74	4.99

In contrast, by aligning to a performance-based requirement, this conservatism can be avoided, resulting in additional design margin and shorter validation time. Design margin benefit is extremely difficult to quantify even on a single I/O type because of the enormous variety of Si circuit designs, fabrication processes, and board designs. Validation time benefit is more straightforward to quantify, however. To empirically confirm that a given link is less than or equal to a certain BER at a certain confidence level, one must test for a sufficient time. The Poisson probability distribution may be used to calculate the required length of test time to validate against a certain BER to a level of 95% confidence, assuming no errors are encountered during the test:

$$Min\_Test\_Time = \frac{-\ln(1-0.95)}{BER \times dataRate}. \tag{2}$$

Table V shows the test time improvement for PCIe Gen 3 and SATA 6 Gb/s using this approach. If an empirical validation test of this nature was implemented in a manufacturing test, for example, this would imply a 99% reduction in test time if it were confirmed that only a BER of 10^{-10} was needed as opposed to 10^{-12}. This is immediately evident from (2): test time is inversely proportional to BER.

One challenge encountered in this study was that, as far as the authors were able to discern, there is no published experimental data of performance penalties vs. BER on modern high-speed interconnects to which a comparison could be made. All previous investigations on this subject appear to be purely theoretical ([7][9]) and did not even analyze a specific existing high speed interconnect type. Because this appears to be an area not previously explored, future studies will include other high speed interconnect types besides PCIe and SATA, as well as other scenarios for PCIe that include a graphics card AIC where hardware acceleration is turned off. It is also the hope that this work will motivate others in the industry to perform studies on their platform architectures.

V. SUMMARY

In this paper, four experiments were conducted to study the impact of increasing levels of BER on performance of high speed serial buses. On a PCIe Gen 3 link running between a CPU and a graphics add-in card, it was found that although there were some card-to-card differences, performance did not start to decrease from error-induced retries until a BER of 10^{-8} at the lowest. On a PCH PCIe Gen 3 uplink to a CPU as well as a SATA 6 Gb/s I/O running from a PCH to a hard disk, performance did not appreciably decline until a BER of 10^{-10} or higher. Finally, the PCIe Gen 3 functional test between a CPU and test add-in card showed that catastrophic performance issues arose at a BER of $\sim 10^{-7}$ but that undetected mismatch errors do not occur until the same level of BER or worse.

The data suggests that many products have additional margin above the 10^{-12} BER spec before any user impact would occur. If new standards and practices were adopted to validate against performance impact instead of a generic BER specification level, conservatism leading to costly over-design and over-validation could be avoided.

ACKNOWLEDGMENT

The authors would like to thank Alejandro Cardenas, Federico Hernandez Reyes, David Steele, and Dale Robbins for performance measurements on the PCH PCIe uplink and SATA. We would also like to thank Tsafrir Waller for the CPU/AIC PCIe performance test runs and analysis.

REFERENCES

[1] PCI Express® Base 3.0 specification, www.pcisig.com [retrieved: Aug, 2014].

[2] IEEE 802.3TM-2012 Section 5, standards.ieee.org [retrieved: Aug, 2014].

[3] Serial ATA Revision 3.1 specification, www.sata-io.org [retrieved: Aug, 2014].

[4] Universal Serial Bus 3.1 Specification, www.usb.org/developers/docs [retrieved: Aug, 2014].

[5] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," Proc. IRE, vol. 49, Jan. 1961, pp. 228-235.

[6] G. Castagnoli, S. Bräuer, and M. Herrmann,"Optimization of cyclic redundancy-check codes with 24 and 32 parity bits," IEEE Transactions on Communications, vol. 41, June 1993, pp. 883-892.

[7] S. Murali, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, L. Benini, and G. De Micheli, "Analysis of error recovery schemes for networks on chips," IEEE Design and Test of Computers, Sep-Oct 2005, pp. 435-442.

[8] W. Liu and W. Lin, "Additive white gaussian noise level estimation in SVD domain for images," IEEE Transactions on Image Processing, vol. 22, pp 872-88.

[9] S. Wang, S. Sheu, H. Lee, T. O, "CPR: A CRC-Based Packet Recovery Mechanism for Wireless Networks," 2013 IEEE Wireless Communications and Networking Conference, April 2013, pp. 321-326.

Functional Testing: A SOA & BPM Approach

William Gontier[1], Franck Hennequin[2], Fabien Lloansi[2]

[1] Cygnus Systems S.A.R.L., Meudon sur Seine, France
[2] SoftAtHome S.A., Nanterre, France

E-mails: william.gontier@cygnus-systems.eu, franck.hennequin@softathome.com, fabien.lloansi@softathome.com

***Abstract*— In this paper, a new approach to automate software reliability verification and validation activities is described. The project has been developed focusing on functional testing of digital television and network home appliances. Nevertheless, at this stage of the project, the developed approach has proven to be sufficiently generic to support a wide range of domains and so it can be of interest for people dealing with functional test automation in a wide industrial range. Most commonly used approaches followed by the industry to automate functional testing require important efforts and skilled human resources in software development to build large sets of specific scripts. Consequently, these approaches cannot be conducted by professionals without programming skills since they are not sufficiently involved in the design, development and maintenance of the tests scenarios. We present in this paper an innovative strategy to overcome the main difficulties. We also show how this new strategy based on a zero-code approach can offer new exciting roles to test team members and deliver an optimized cost structure of test automation activities.**

Keywords-Functional Testing; Test Automation; Zero Code; Cost Optimization; Service Oriented Architecture; Business Process Modeling.

I. INTRODUCTION

The software development activities are each day more "test driven". The main reasons for such a trend mainly lie in the increasing complexity of the developed systems, the integration of third-party software modules (commercial and open source), the interactions with external systems partially mastered, the pressure put on the R&D teams with respect to constraints of "time to market", the difficulties in getting clear, complete and detailed specifications before the project starts, the widespread and success of agile methods which significantly helped software organizations detect the benefits of test driven development methodologies.

As an immediate consequence, functional testing is each day a more strategic step in the product development cycle but it is also a more costly activity for software organizations due to the increasing number of tests required to deliver adequate test coverage of products.

Thus, poor testing coverage and/or inadequate test automation strategies and/or inadequate testing tools can prove to be detrimental to the competitiveness in terms of product quality and cost.

In this context, automating functional tests becomes each day a more challenging topic for software organizations. This paper reviews the commonly followed approach in the industry to carry out test execution and automation, for which the creation of scripts is based on coding, possibly simplified with an interface to abstract this layer. Identifying the key weaknesses of these approaches, we present an innovative strategy to overcome the main difficulties encountered in automating functional test activities. We show how this new strategy improves the quality of execution during functional test campaigns while providing an optimized cost structure.

Beyond this introduction, Section 3 presents the most frequently encountered test execution and automation strategies in the industry to compare them with the approach described in this paper. We also introduce the Saturn software framework implementing this new strategy in the following section. Saturn is currently used for functional testing of products developed by the company, SoftAtHome, including its digital television receivers/decoders and xDSL/fiber Internet gateways. Section 4 deals with the main results delivered by Saturn. Finally, before concluding, Section 5 introduces the evolutions of Saturn that are currently under development.

II. State-of-the-art

An efficient and fully automated system must combine an easy interface to generate automation tests (no coding skill required) and the possibility for users to add their own modules (new functionalities or devices).

Main actors in automated testing proposed solutions based on scripting that requires lengthy and expensive coding phases (often in python language). In order to bring more flexibility for the user, these solutions can include a package of additional libraries (for example, *RT-RK* company anticipated from future users include list of libraries [7]). These solutions require coding skills that often differ from tester to tester, according to their profiles.

To abstract the coding layer, they subsequently developed additional interface (drag and drop). User can access to the toolbox that simplifies scripts developments. However these glue layers are fully linked with the automation tool and the user of this tool is dependent on these solution concept companies to supply him with future evolutions of their product (sometimes the solution needs a proprietary language, as *StormTest* [8], the solution developed by *S3 Group*). Consequently, there is no more versatility for the user to generate evolutions.

Our new approach is to directly start with a modeling system (BPM) that can interface with a toolbox suitable. In others words, to use a system adapted for sequencing and add application layers used through independent connectors.

III. Strategies for Test Automation

A. *Test Execution & Automation **methods***

Manual testing generally delivers imprecise test results and fails to reproduce tests, mainly because it is largely subject to the interpretation of a human operator whose judgment is more likely to evolve over time.

On the other hand, test campaigns using robots can significantly improve the stability of test procedure results and reproduction over time (which is essential, for example, in the case of tests performed within the context of certification processes).

Nevertheless, it is quite frequent to see test automation reduced to its simplest form, and thus, limited to the development of scripts specifically developed for the product to be tested. Such scripts are sometimes even developed by the teams who have contributed themselves to the development of the tested product.

Many experiences show that this approach generally gives poor results. Indeed, if a test case can be decomposed as a succession of steps to execute, this structure can't be kept with a script in a coding language. Consequently, the understanding of an automation script will required a code review.

Furthermore, this bias, associated with the implementation of complex tests for which developers in charge may lack time and/or skills to implement the complex algorithms required to replace human skills (e.g., computer vision), leads the tests implementation to be based, most of the time, on optimistic use cases.

In addition, some defects resulting from omissions committed during the design and / or development will typically be perpetuated when the product and the test scripts are done or described by the same developers. Thus, using the common approach, the automation process leads both to replace a human operator by an automatic operator, but also to adapt the tests to be implemented within a reasonable time frame and since its use requires programming skills, thus needing a engineering profile different from that of an expert in validation.

Consequently, with the common approach to test automation, significant bias can be introduced, typically leading to degraded functional tests coverage.

Just the opposite, the approach followed by the Saturn project (modeling system BPM) ensures that the test team - whose role is to systematically search for defects of any kind in the products they have in charge - retains its prerogatives during the design, the development, the maintenance and the execution of the test campaigns.

To achieve this goal, Saturn delivers to test team members a suitable toolbox to deal with all the activities related to test scenario management but without requiring expertise in the field of computer programming.

Furthermore, the Saturn toolkit provides the tools necessary to replace human capacities in the field of testing activities. This includes, for example, computer vision algorithms, image quality assessment (taking into account the human visual system), identification of soundtracks, and more others features.

In addition, Saturn tools can provide valuable information about the diagnosis of the system due to their inherent ability to quickly process large amounts of data (e.g., protocol analysis).

B. *Costs of Test Execution & Automation*

While manual testing triggers operating expenses (OPEX) without offering any possibility of cost sharing on the number of test campaigns to be achieved on time, the development of tests automation tools constitutes an investment opportunity (CAPEX).

It is important to note that compared to the traditional automation approach (i.e., specific scripts development), the approach proposed by Saturn, which consists in offering a generic and reusable toolbox, can generate some revenue. Indeed, generic tools can be marketed and therefore likely, to attract customers and business partners in a given industry. These business opportunities can help finance investments in tests automation through income generation. Furthermore, the development of generic tools is susceptible to help in extending the amortization period of the developments due to a longer depreciation period thanks to a better sustainability over time of the developed tools. All these aspects contribute to minimize overall labor costs since it spares using engineers for the development of automatic tests. In the Saturn approach, only widely reused generic tools require contribution from specialized software developers while as automated tests are implemented by technicians.

These considerations, summarized by in Figure 1, make the strategy deployed through Saturn quite an optimized strategy in terms of TCO (Total Cost of Ownership) of an automated test infrastructure.

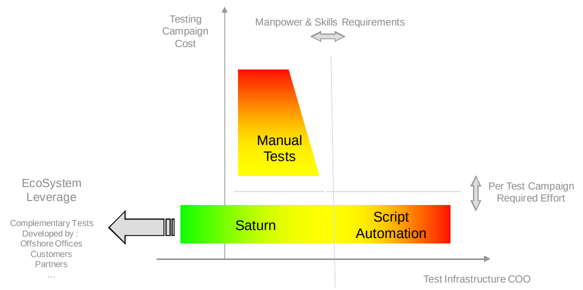


Figure 1. Cost structures of manual testing vs. script developments vs. Saturn approaches.

IV. Implementation

To enable test team members to develop their test scenario without requiring computer programming skills, the Saturn system had to provide a way to describe tests procedures in a graphical form.

A. The Business Process Modeling Approach

The approach consisted in selecting the BPMN (Business Process Modeling Notation) language in its 2.0 version [6]. This language offered all the key characteristics for the development and the maintenance of test scenarios, as well as both easy to learn and intuitive to use.

With our approach, a test scenario is developed as a BPMN process made of connected activities (cf. Figure 2). Each connection can activate - at the next execution step - an activity if it is connected to the currently active activity and the condition associated to the connection is evaluated to TRUE at execution time. An activity can execute a sub-process, perform some local actions such as updating local or global variable values, or call connectors. Connectors are typically predefined routines performing some frequently required tasks. In the case of Saturn, the connectors are employed to access the toolbox API (Application Programming Interface). Each Saturn connector implements a web service call to a wrapper delivering a specific service (e.g., checking the presence of a given pattern on the television screen thanks to the computer vision wrapper managing video acquisition hardware). Connectors are used by the scenario developer as a way to access services delivered by the wrappers of the Saturn framework and so, interacting with the external devices to be tested. The state of the tested device is known from the test scenario thanks to the results returned by the wrapper calls.

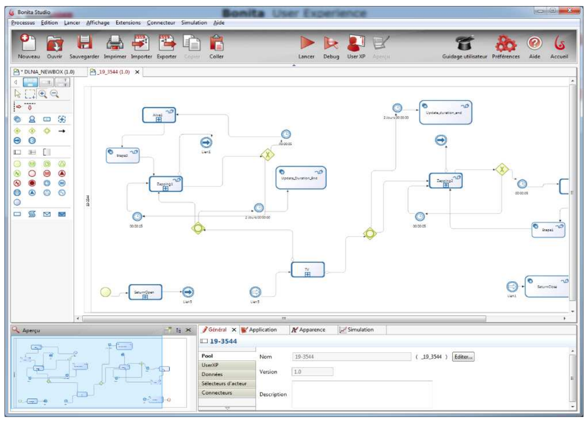


Figure 2. Test scenario example developed in BPMN 2.0 with the Bonita Studio editor. Thanks to the BonitaSoft solution the BPMN 2.0 files are then migrated to the Java framework and executed on a JBOSS server.

B. The Wrappers

The wrappers are server applications hosted by the tests robots whose role is to provide the services required to test scenarios for what concerns specialized functions typically interacting with the physical world (e.g., pattern detection on a television screen).

Wrappers typically incorporate SDK (Software Development Kit) to manage the robots' hardware components (e.g., video capture card) and / or specific algorithms (e.g., audio identification algorithms). A description of the main algorithms developed for the project can be found in [1][2][3][4].

With our approach, the BPMS (Business Process Modeling System) acts as a sequencer calling - through Java connectors - the services rendered by the wrappers (Figure 3). Communication between the BPMS and the wrappers is done via a session oriented API whose structure is common to all the wrappers. At this date, the main Saturn wrappers are:

- Vision (shown in Figure 4) deals with the computer Vision algorithms toolbox such as pattern matching, video detection, screenshots, optical character recognition, etc.
- Audio dealing with audio processing services such as audio watermarking, audio detection, audio track identification, etc.
  Audio contents are tagged using different amplitude modulation (cf. Figure 5)
- Studio (shown at Figure 6) dealing with audio/video content management services such as stream generation, video frames identification, video quality assessment (PSNR, SRSIM, etc.), "lip synching" computation.
- Web UI dealing with Internet browser control used for web user interface and web services testing.
- Power dealing with external devices power supply management services.
- RCU dealing with remote control unit services, such as infrared and radio frequency based remote control simulators.
- Traces dealing with equipments traces and logs management services.

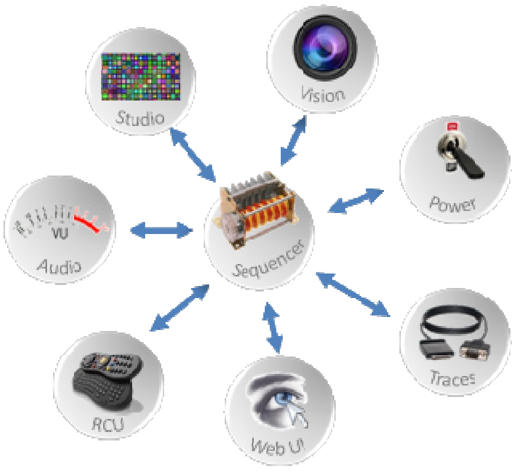
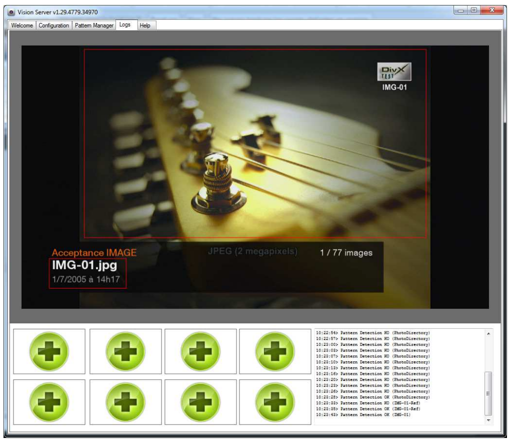


Figure 3. SOA (Service Oriented Architecture) illustrated: test scenarios executed by the BPMS (Business Process Modeling System) server interact through a standardized web services API with the wrappers applications hosted by the robots.



```
10:22:54> Pattern Detection KO (PhotoDirectory)
10:22:57> Pattern Detection KO (PhotoDirectory)
10:23:00> Pattern Detection KO (PhotoDirectory)
10:23:03> Pattern Detection KO (PhotoDirectory)
10:23:07> Pattern Detection KO (PhotoDirectory)
10:23:10> Pattern Detection KO (PhotoDirectory)
10:23:13> Pattern Detection KO (PhotoDirectory)
10:23:16> Pattern Detection KO (PhotoDirectory)
10:23:20> Pattern Detection KO (PhotoDirectory)
10:23:23> Pattern Detection KO (PhotoDirectory)
10:23:26> Pattern Detection KO (PhotoDirectory)
10:23:28> Pattern Detection OK (PhotoDirectory)
10:23:33> Pattern Detection KO (IMG-01-Ref)
10:23:35> Pattern Detection OK (IMG-01-Ref)
10:23:43> Pattern Detection OK (IMG-01)
```

Figure 4. Saturn powerful computer vision system. Example of patterns recognition and localization on a set top box video output.

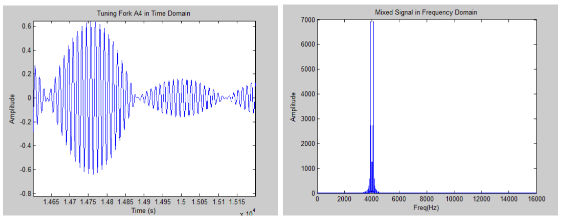


Figure 5. Audio watermarking by amplitude modulation (resp. time and frequency domains).

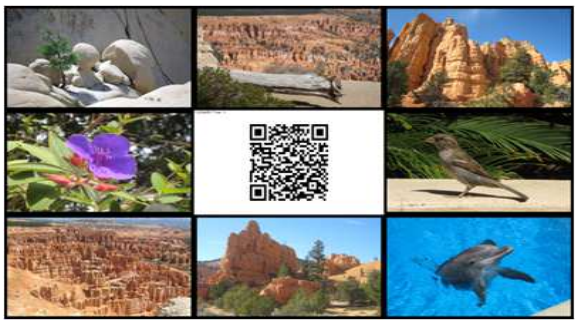


Figure 6. Video frame identification and clock synchronization by QR Code insertion/decoding.

C. *The Catcher Application*

As shown in Figure 7, the catcher application, allowing the test execution infrastructure to communicate with the information system in charge of the test plans and test results management, plays a central role in Saturn.

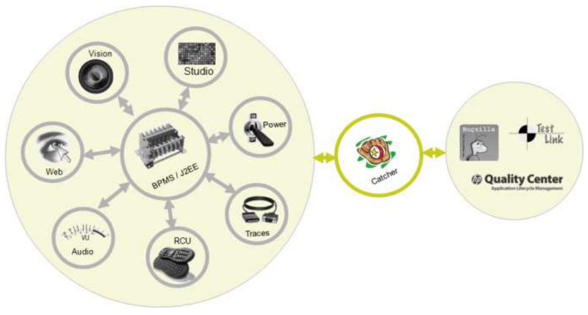
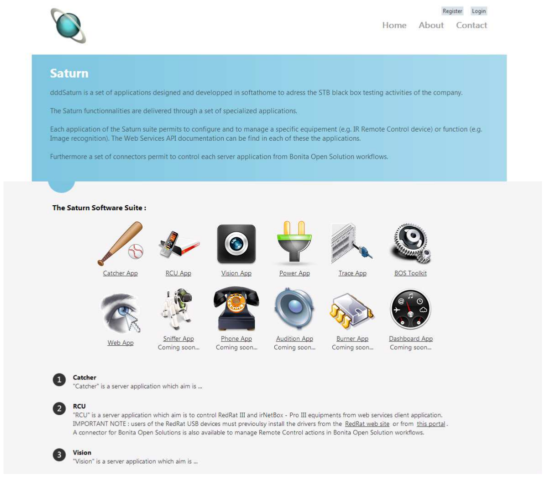


Figure 7. The "catcher" application in the system's architecture.

The Catcher application main roles consist in extracting the test campaigns descriptions stored in a third party application of a test management system (for example, TestLink, HP Quality Center), in presenting tests scenarios to the operator in different levels of aggregation such as unit testing, test sequence or functional modules, in deploying the Java code corresponding to the tests to be executed by the JBOSS server, in controlling the execution of the test scenarii (e.g., abort a test sequence in case of critical error), in collecting traces obtained during tests execution, in attaching these traces and various additional information (e.g., TV screen captures) to the test reports, in posting the test results to Testlink, HP Quality Center or any test management third party application, in keeping track of files versions, in managing the files versioning system (as *Git* [11]) and in centralizing the Saturn's configuration parameters.

D. *The Saturn Portal*

The Saturn web portal shown in Figure 8 has been developed to provide a single point of access to system's users. It mainly hosts: the wrappers applications providing versioned automatic updates of the applications installed on the robots, the files repositories for multi-sites deployments (tests scenarios written in BPMN 2.0, Java classes, logs, screenshots, test traces, A/V streams, patterns, etc.), the wrappers' databases, the Saturn portal and the user documentation.



Saturn

dddSaturn is a set of applications designed and developped in softathome to adress the STB black box testing activities of the company.

The Saturn functionnalities are delivered through a set of specialized applications.

Each application of the Saturn suite permits to configure and to manage a specific equipement (e.g. IR Remote Control device) or function (e.g. Image recognition). The Web Services API documentation can be find in each of these the applications.

Furthermore a set of connectors permit to control each server application from Bonita Open Solution workflows.

The Saturn Software Suite :

1 **Catcher**
"Catcher" is a server application which aim is ...

2 **RCU**
"RCU" is a server application which aim is to control RedRat III and irNetBox - Pro III equipments from web services client application.
IMPORTANT NOTE : users of the RedRat USB devices must previoulsy install the drivers from the RedRat web site or from this portal .
A connector for Bonita Open Solutions is also available to manage Remote Control actions in Bonita Open Solution workflows.

3 **Vision**
"Vision" is a server application which aim is ...

Figure 8. Screenshot of the Saturn web portal.

V. MAIN RESULTS

A. *Saturn Key Benefits*

Among the main benefits experienced with the Saturn solution deployments, can be mentioned: a clear separation of the R&D and test team roles, the increased motivation of the test team members in dealing with the whole test strategy and not only with repetitive task execution, an increased autonomy of test teams in the test automation process, fully automated process supported by an easy integration of test management tools (TestLink [9], HP Quality Center [10]) dealing with IT and reporting tasks automation, integrated test scenarios versioning and reviewing process (using the version control system GIT), easy reuse of already developed test scenarios via BPMN 2.0 sub-processes mechanism, version tracking of the totality of the elements involved in test results, quality of test results related to the reproduction of the test runs, multi-site robots deployment with file versioning and repositories synchronization, limited training required for newcomers thanks to easy use of intuitive tools, scalable and easy to maintain architecture with new testing requirements managed by adding separate wrappers and connectors in an incremental approach without any impact on the existing system.

B. *Quantitative Results*

The Saturn test automation framework implements this strategy and is used by the company SoftAtHome, in the implementation of its automated testing infrastructure for "Set Top Boxes" and "Home Gateway". Saturn is currently deployed in 4 countries: France, Belgium, UAE and Tunisia with about 20 test robots connected through the internet to a shared infrastructure. One third of the manual validation can be executed with automation system Saturn. Each month, more than 6 middleware releases are tested (robustness and no regression campaign).

VI. CONCLUSION

We presented a novel approach to deal with functional tests automation. This approach - built around a Business Process Modeling System and according to a Service Oriented Architecture - instead of targeting specific scripts development for tests automation - focuses on the delivery of a generic toolkit which aims to deliver a set of human replacement tools that can be used by testers without programming skills.

The versatility to add new wrappers brings many perspectives for Saturn tool in particular to interface with additional devices useful for Set Top Boxes validations: EDID Extended Display Identifier Data generator (as *Quantum* [12], a EDID generator that can simulate a connection with all kind of TV sets), stream player (as *DekTek* modulator [13] able to broadcast a specific stream content mandatory for the automated test), etc.

Moreover, a new Saturn wrapper offering innovative IP Network datagram analysis services [5] is currently under development. The main goal is to offer a toolbox to develop automation test cases for Home Gateway (basic network) and to check the interoperability with Set Top Boxes, by the way of common scripts.

ACKNOWLEDGMENTS

The Saturn project is financially supported by the company SoftAtHome. SoftAtHome leverages the best of connectivity, Pay TV and Digital Home Services into SOP, its unified software platform for Set Top Boxes and Home Gateways.

The main contributors to the Saturn project development are in alphabetical order: Mr. Christian Couder, Mr. Vincent Courtot, Mr. Olivier Delfosse, Mr. William Gontier, Mr. Franck Hennequin, Ms. Alison Jorre, Mr. Saddek Kadji, Mr. Fabien Lloansi, and Mr. Marc Toffanin.

REFERENCES

[1] William Gontier, Franck Hennequin and Marc Toffanin, "Automated Detection of Video Artefacts", SoftAtHome S.A., September 2013 (see condition 1).

[2] William Gontier, Franck Hennequin and Marc Toffanin, "Audio watermarking and Identification Algorithms", SoftAtHome S.A., September 2013 (see condition 1).

[3] William. Gontier, Franck Hennequin and Marc Toffanin, "Lip Sinching computation in Saturn Test Automation Framework", SoftAtHome S.A., November 2013. (see condition 1)

[4] William Gontier, " RCU Simulator SDK Specifications - Saturn Framework", SoftAtHome S.A., March 2014 (see condition 1).

[5] William Gontier, "IP Network Test Automation - A big Data Approach", SoftAtHome S.A., May 2014 (see condition 1).

[6] Robert Shapiro and Stephen A. White, "BPMN 2.0 Handbook Second Edition: Methods, Concepts, Case Studies and Standards in Business Process Modeling Notation (BPMN)", Layna Fischer, 2012.

[7] http://bbt.rt-rk.com/en/test-suites

[8] http://www.s3group.com/tv-technology/services/stormtestr-test-services/

[9] TestLink Open Source Test Management https://wiki.openoffice.org/w/images/1/1b/Testlink_user_manual.pdf

[10] Quality Center Enterprise http://www8.hp.com/us/en/software-solutions/quality-center-quality-management/

[11] Git http://git-scm.com/

[12] http://quantumdata.com

[13] http://www.dektec.com/ /

(1) Delivered subject to non disclosure agreement.

Contact SoftAtHome : http://www.softathome.com/pages/contact

A Model-Based Testing Methodology for the Systematic Validation of Highly Configurable Cyber-Physical Systems

Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria

Computer and Electronics department
Mondragon Goi Eskola Politeknikoa
Goiru 2, Mondragón
Email: {aarrieta, gsagardui, letxeberria}@mondragon.edu

***Abstract*—Nowadays, the society is dependent on Cyber-Physical Systems (CPSs), which are complex systems that combine digital technologies and physical processes. The need for dealing with constant changes in products is leading these systems to handle variability in several aspects, which entails to a considerable increase in the complexity of the systems. Many of the research efforts are focused on the efficient development of these systems. Nevertheless, the infeasibility of testing all the possible configurations, the unclear notion of the achieved test coverage and the high amount of time required make testing processes non-systematic and challenging. This paper introduces the main problems for testing highly configurable CPSs and proposes a novel approach for testing systematically and efficiently while achieving high test coverage.**

Keywords–Model Based Testing; Test Methodology; Variability Modelling; Cyber-Physical Systems

I. Introduction

CPSs integrate digital cyber computations with, often complex physical processes, where embedded systems monitor and control physical processes with sensors and actuators [1]. The dependency of the society on CPSs that control many individual systems and complicated coordination of those systems is considerably increasing [2]. CPSs working in industries are highly complex systems [2], which make embedded systems to come with a set of configuration parameters [3]. As a consequence, the variability of CPSs increases, and the embedded systems have to deal with the changes that the physical environment requires.

Variability is the ability to change or customize a system [4], also understood as configurability (variability in the product space) or modifiability (variability in time) [5]. CPSs handling variability are commonly known as highly configurable CPSs, which are described as heterogeneous systems where hardware and software are integrated with the aim of controlling a physical process. These systems share the same embedded software code base, which has the ability of getting configured to work in systems with different features [6]. This configurability might be realized by parameters, where changes in the configuration of the product might lead to a complete different system's behaviour [7].

Testing highly configurable CPSs is a challenging task. These kind of systems can get configured into thousands or even millions of configurations, which makes it infeasible to test every single configuration and as a consequence the notion of the achieved test coverage is uncertain for quality engineers. Thus, ensuring that the CPS will meet all requirements in every possible configuration is not realistic.

Two main challenges are addressed when testing highly configurable CPSs. Each test can be executed many times, once for each possible configuration, and as a result, "the cost of running a test suite is proportional to the number of tests times configurations" [8]. Selecting concrete configurations as well as the right test cases for the selected configurations is one of the principal challenges when testing highly configurable CPSs.

On the other hand, the use of Model-Based Design (MBD) tools such as MATLAB/Simulink is increasingly growing when designing and testing CPSs. The high number of variants and the complexity of the CPSs make manual configuration error-prone and inefficient, which warrants the need for an automated solution for the configuration of CPSs [2], as well as for the test system, where a configurable test architecture handling variability becomes essential.

Section II of this paper introduces the related work. The proposed approach for the systematic validation of highly configurable CPSs using a model-based testing methodology is presented in Section III. Some discussions of the proposed approach and expected contributions are discussed in Section IV. Section V describes preliminary and expected results. Finally, Section VI presents conclusions and future work.

II. Related Work

The work presented by Bauer et al. [3] proposes a systematic model-based test approach named REDUCE for the validation of highly configurable safety-critical systems. This approach uses combinatorial and model-based technologies with the main objective of reducing configurations and test cases until reliability estimations based on testing become feasible. The approach presented in [3] is focused on the reduction of configurations and possible test cases for the validation of highly configurable safety-critical systems using model-based statistical testing, while our approach will be more focused on the management and test architecture for the validation of CPSs. In addition, in the approach presented in [3], the test model is created for each system configuration, reusing similarities between different configuration-specific test models. In our approach, a test model handling

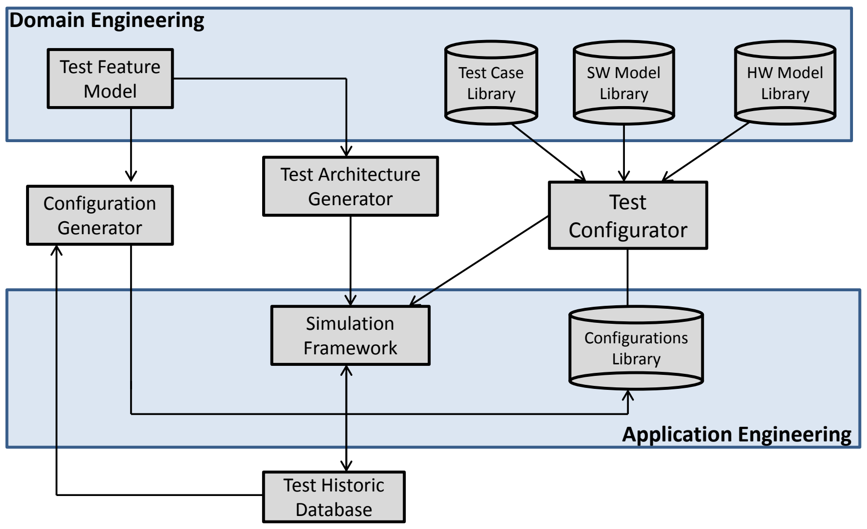


Figure 1. Overview of the proposed model-based methodology for the systematic validation of highly configurable CPSs.

variability in the test architecture as well as in the test cases is proposed, as the study presented by Weißleder and Lackner [9] demonstrates that it is more efficient binding variability after test case design.

A similar approach for highly configurable embedded systems in the automation domain is proposed by Streitferdt et al. [7]. In this case, a testing process is developed, where test cases are automatically generated with a parameter model in combination with a test model.

Combinatorial Interaction Testing (CIT) techniques are also widely used when testing configurable systems, where configurations are often selected using pairwise or t-wise techniques. According to Kuhn et al. [10], testing efficiency using pairwise testing method is 2.4 times higher and quality a 13 % better than manual testing method. CIT techniques are commonly combined with model-based testing as proposed by Oster et al. [11], where a tool chain is introduced named MoSo-PoLiTe. This tool selects pairwise configuration selection component on the basis of a feature model covering 100% pairwise interaction, and test cases are generated for each configuration.

Nevertheless, these combinatorial techniques select product configurations statically. In [8], a novel approach is presented named SPLat, where product configurations are determined dynamically during test execution by monitoring accesses to configuration variables. The technique consists in executing the test for one configuration, while observing the values of configuration variables used to prune other configurations.

Another approach to reduce validation costs of highly configurable systems is minimizing the test suite for testing a product, reducing redundant test cases. A set of test cases can be automatically obtained by selecting features of a feature model to test a new product, but there still can exist redundant test cases [12]. A fitness function based on three key factors (Test Minimization, Feature Pairwise Coverage and Fault Detection Capability) for three different weight-based genetic algorithms is defined in [12]. This approach allows reducing the test suite covering all testing functionalities achieving a high fault detection capability.

From the testing perspective, our previous work [13] presents an approach based on model-based testing and variability management integrated in Simulink, where a concrete configuration of the software is chosen by the test engineer, and the testing infrastructure is instantiated for the chosen configuration. In another previous work [14], a testing architecture with variability management in the test oracles is proposed to test distributed robotic systems in Simulink. A product line of validation environments with variability to test different applications in different domains and technologies is proposed by Magro et al. [15]. However, these works do not consider the automatic generation and configuration of the test architectures, there are not oriented for highly configurable CPSs, thus, they do not consider test case selection and test suite minimization.

Model-in-the-Loop for Embedded System Test (MiLEST) is a toolbox for MATLAB/Simulink developed by Zander-Nowicka in [16]. This test architecture is oriented for the validation of automotive real-time embedded systems in Model-in-the-Loop (MiL) phase. The main advantages of MiLEST is that it is oriented for the automotive domain. This industry is the one that most uses variability modelling according to [17]. Another advantage of MiLEST is the compatibility with Simulink, which is a simulation tool widely used to model embedded software and simulate CPSs. The test architecture used in this methodology will be based on MiLEST but it will address some changes: The developed architecture will handle variability issues and will be able of automatically getting configured. In addition, it will contain algorithms for the efficient validation of highly configurable CPSs. In addition, the test architecture will be communicated with a database with test historic data in order to prioritize test cases to be executed. Finally, the test architecture is expected to be used not only in MiL phase, but also in software, processor and hardware-in-the-loop phases.

III. Approach

The methodology begins from a manual implementation of a Feature Model that manages the variability of both, the Cyber-Physical System Under Test (CPSUT) and the test architecture. The feature model is developed using the tool FeatureIDE [18], which automatically generates a .xml file. This .xml file will be used for the automatic generation of the test architecture and CPSUT Simulink model.

Once the model is automatically generated, the first configurations will be generated by the configuration generator. These configurations will be used by the test configurator to configure the simulation framework. Once configured the simulation framework, the test will be run and the results uploaded to the test historic database. Taking these results into account, the configuration generator will generate other configurations.

Figure 1 depicts the proposed methodology. Its components are classified following the theory of product lines: Domain Engineering and Application Engineering. The components in the domain engineering are for the whole product line, what means that variability is implemented. On the contrary, the components of the application engineering side correspond to a concrete product configuration, thus, variability is already bound.

A. *Feature Model*

According to Berger et al. [17], Feature Models are the most used notation in order to model variability of systems in industrial practice. As mentioned before, we propose using the tool FeatureIDE [18]. The main reason for using FeatureIDE is its simplicity, its completeness and the availability of its source code, which enables adapting the tool to our needs. In addition, this tool is constantly under construction, with new updates becoming available. It also uses CIT algorithms to generate correct configurations using pair-wise or t-wise techniques, which can be reused when implementing our dynamic configuration generator.

Figure 2 depicts a basic feature model from the mobile industry, where the basic functions of a feature model are shown. The features allow the following relationships: “mandatory”, “optional”, “alternative” and “or”. In addition, constraints between features can be specified by the relationships “requires” and “excludes”.

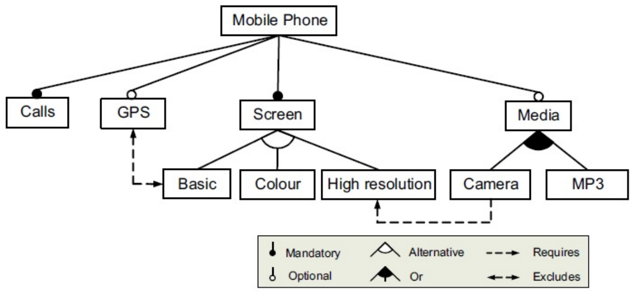


Figure 2. Example of a Feature Model from the Mobile Industry [19].

As shown in Figure 3, the feature model is composed of the CPS Feature Model, the Test Feature Model and the Integrator. The CPS Feature Model is the feature model related to the highly configurable CPS, which manages the variability that the CPS has. The Test Feature Model is related to the test architecture, and manages the variability of the test architecture. Lastly, the integrator is a file that enables the integration of all the components of the model among them, which is used to automatically generate the Simulink model of the CPSUT and the test architecture.

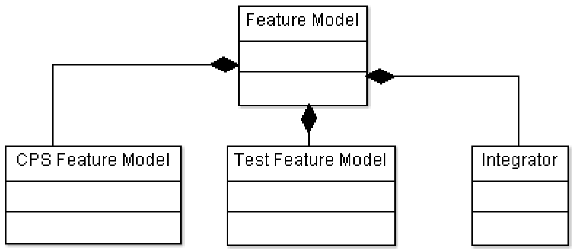


Figure 3. Meta-model of the Feature Model proposed for the systematic validation of highly configurable CPSs.

Figure 4 shows an example of how a feature model could be. The motivating example illustrates the control of the liquid of an industrial tank. The liquid can be a chemical product or water. In the case the liquid is a chemical product, the pH must be measured by a pH sensor. In addition, an optional temperature sensor can be used to measure the liquid’s temperature. The components of the CPSUT have been placed on the right branch of the feature model. On the left side, the variability of the test architecture is modelled. Traceability between both sides is modelled with constraints, as shown in Figure 4 (“requires”).

B. *Test Architecture Generator*

Once the feature model is built, FeatureIDE generates a .xml file which is read, together with the integrator file by the test architecture generator, which is implemented in MATLAB to automatically generate the Simulink model with the CPSUT and the test architecture. The main work of the integrator is to handle information about connection among the ports of the components’ models. The Simulink model is automatically generated using MATLAB scripts. In addition, the comonents’ models (sensors, actuators, etc.) will be designed by system and test engineers and saved into a Simulink library. Later, this library is going to be used when automatically configuring the Simulink model (as shown in Figure 5).

C. *Configuration generator*

One of the key points when testing highly configurable systems is to efficiently generate configurations. Current studies describe different static CIT techniques to generate configurations, e.g., [3][10][11]. FeatureIDE [18] also allows generating product configurations both automatically and manually, and these configurations are saved into a .config file.

When testing highly configurable systems, product configurations can be generated statically or dynamically. Statically

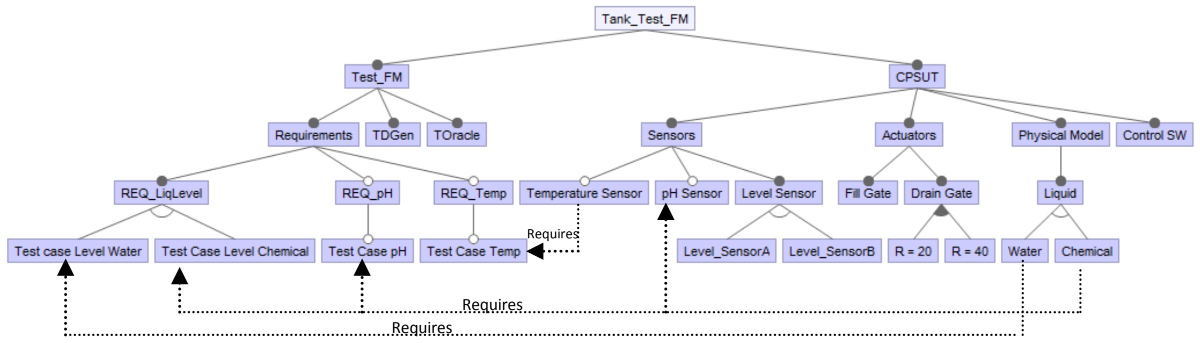


Figure 4. Example of how the Test Feature Model looks like.

generating configurations means generating a set of configurations and later testing those configurations. On the contrary, configurations are generated dynamically when the configuration generator algorithm generates a product configuration, the configuration is tested and depending on the test results and objectives the configuration generator generates another product configuration, e.g., [8].

Our hypothesis is that generating configurations dynamically is more efficient than generating them statically, as test results might have an influence when generating configurations. For instance, taking the before explained tank liquid controller example, the temperature sensor might be giving errors in the system. When generating configurations, the configuration generator should take into account this issue and generate configurations including the temperature sensor. Our configuration generator will be developed in C++, and it will communicate with both, the feature model (in order to obtain correct configurations) and the test historics database (to obtain information of the executed tests, test quality, etc.). The generated configurations are saved in a file with a .config extension and stored in the configurations library.

In order to generate configurations dynamically, it is necessary to study test quality metrics related to highly configurable CPSs. These metrics will be related to requirements coverage, features coverage, components coverage, etc.

D. Test Configurator

The configuration file is read by the test configurator, which will be implemented in MATLAB and whose main task is to automatically configure the CPSUT and the test system in a Simulink model.

To achieve this goal, the components corresponding to the CPS are allocated into a Simulink library. The previously explained test architecture generator automatically generates a first model, which consists of the principal components (Taking Figure 5 as example, Sensor1, Sensor2, Actuator1, Actuator2, REQ1 and REQ2). When the configuration is parsed by the test configurator, the principal features are replaced by the components corresponding to the CPS configuration, i.e., sensors, actuators, etc.

When modelling variability in Simulink, two kind of models can be used: 100 % models and 150 % models. The 100 % models are the models that allocate just the components corresponding to a concrete configuration. For example, first class variability modelling technique can be used to model variability this way, as proposed by Haber et al. [20]. On the contrary, the 150 % models allocate all the components in the models, the components related to a concrete configuration are selected with Simulink blocks such as switch or merged, e.g., [21] [22]. The main drawback of 150 % models is that as there are components not selected for a configuration allocated in the Simulink model, the simulation time is not optimal, which increases the overall validation time. Due to this reason, we do not rely on 150 % models, and our approach is designed to use 100 % models and optimize the simulation time.

E. Test Architecture

The test architecture is essential in order to carry out a systematic validation of any system and reuse the test cases along the whole verification and validation process. As mentioned before, the test architecture is going to be an adaptation of MiLEST [16], a test architecture developed to test real-time embedded systems of the automotive domain. The hierarchy of MiLEST is divided into four abstraction levels: Test Harness level, Test Requirement level, Test Case level and Feature level. The main components of the test architecture are shown in the meta-model depicted in Figure 6.

MiLEST will be adapted to be configurable and be able of testing any configuration of the CPSUT. To achieve this goal, its components will handle variability. Variability in the test data generator will be found in signals (number and characteristics of each signal), requirements (number and parameters of the requirements), test cases (test case duration and test case characteristics), etc. In the test oracle, variability might be found in signals (number of input signal to the oracle), requirements (number of requirements, number of validation function characteristics, parameters), etc.

In addition, a communication with a test historic database will be implemented to execute test cases according to the previous results. The test controller will encapsulate genetic algorithms with the objective of minimizing the test suite and prioritizing test cases.

Figure 6 shows the composition of the test architecture and the communication among its components. The test architecture is composed of the test data generator, the test controller

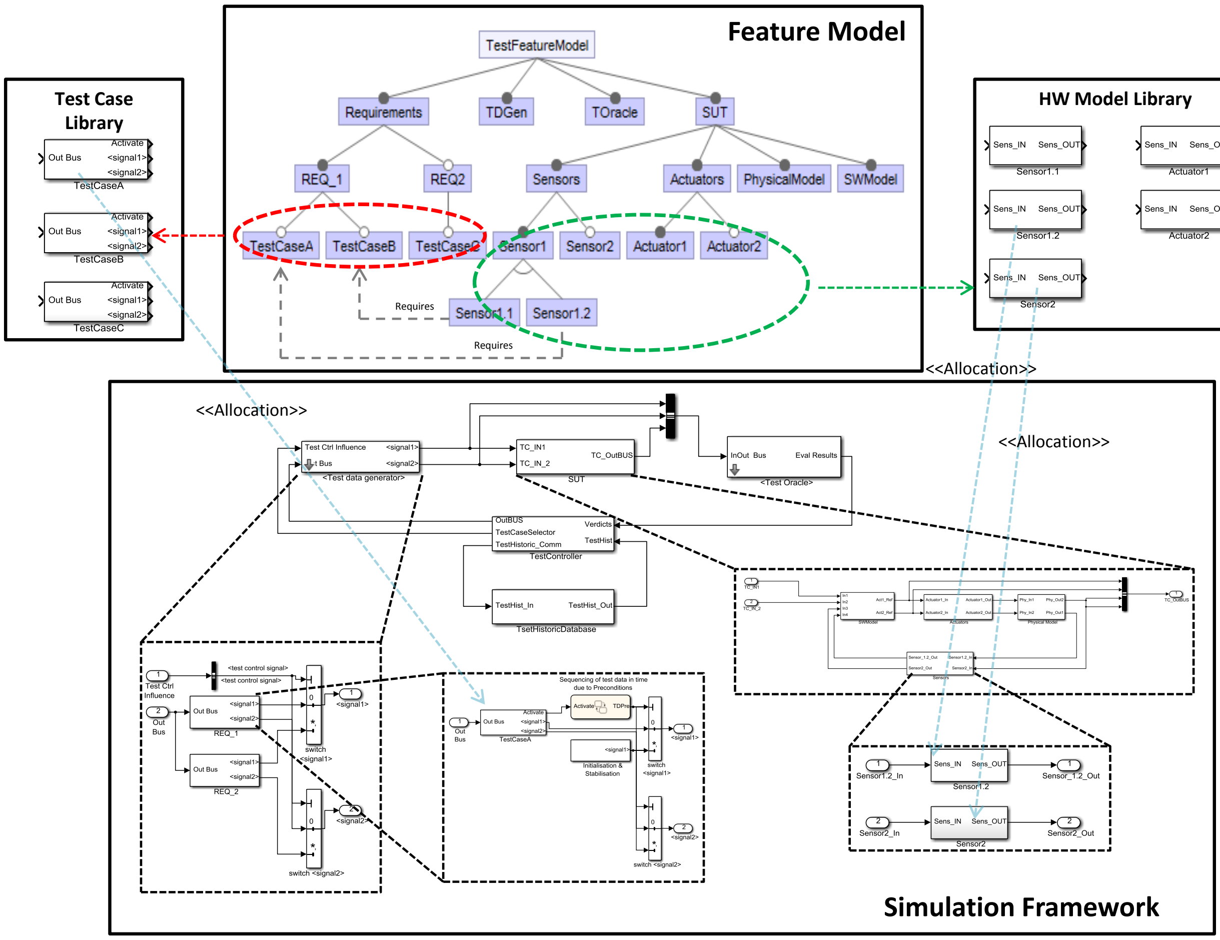


Figure 5. Overview of the relations among the features in Feature Model and Simulink Model.

and the test oracle. The test data generator is the source in charge of stimulating the CPSUT executing test cases. Test cases are implemented manually by test engineers, and each test case will test a single requirement of the system.

With regard to the test controller, it will select one test case or another depending on the test purpose; before executing any test, the test controller will obtain information about the previously executed test cases by communicating with the test historics database.

Finally, the test oracle will evaluate whether the expected behaviour of the CPSUT is correct or not. The test oracle will have one sub-oracle for each requirement, and each sub-oracle will be composed of one or more precondition and assertion, which are manually implemented by the test engineer, taking into account variability.

F. Cyber-Physical System Under Test

Figure 7 shows the system to be tested, where the components are differentiated into Cyber and Physical. In our approach, the CPSUT is kept as black-box, where the test engineer does not need to be familiar with its internal behaviour

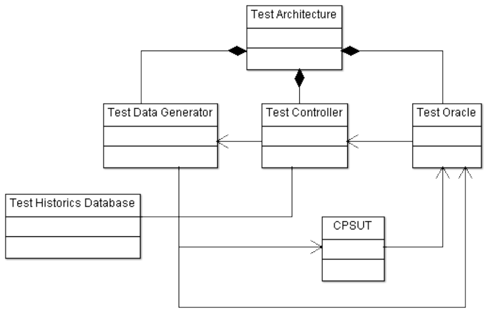


Figure 6. Meta-model of the test architecture.

[23]. Variability in a CPS can be found in the cyber as well as in the physical side. Variability in the physical side, commonly known as context variability, is related to the variability of the environment, i.e., the number and characteristics of sensors and actuators, variability of the mechanics, etc. The embedded system also has to handle variability in order to deal with the

variability of the physical side. Variability of the cyber side can be related to the software, where different configurations are achieved depending on the configuration of the physical side, or to the hardware (types of microprocessors to be used, etc.).

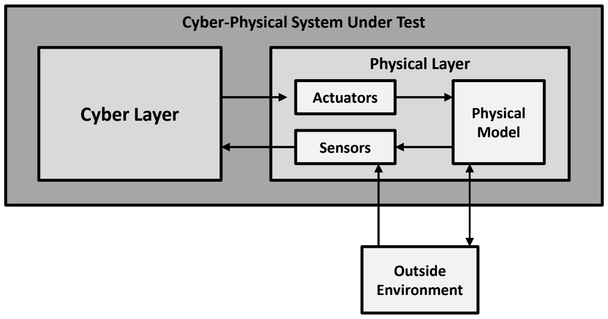


Figure 7. High level overview of the main components of a CPS.

Model-Based Design (MBD) tools are used for the development as well as for the testing of CPSs. Model-, Software-, Processor- and Hardware-in-the-Loop (MiL, SiL, PiL and HiL) tests, provide four testing phases [24], which are typically used to test CPSs in different stages and testing objectives. Our approach will study the possibility of testing the CPSUT in these four stages automatically.

Figure 7 also depicts a block named outside environment. This block refers to the environment in which the CPS resides, which often has a strong influence into the behaviour of the CPS. Considering the example of a mobile robot, the outside environment would include the obstacles to which it is exposed, e.g., the slopes, the surface or even the temperature or humidity which can lead to an inadequate performance of the on-board electronics.

IV. Discussion

The goal of the proposed methodology is to systematically validate highly configurable CPSs. To achieve that goal it is necessary to obtain the needed configurations and test cases to ensure the correctness of highly configurable CPSs in any possible product configuration. The main contributions of the study are foreseen to be the following:

- A methodology to systematically manage the validation of highly configurable CPSs. This methodology begins with the implementation of a feature model. A feature model is a notation that represents the features and relations among them of all possible products of a Product Line represented as a hierarchically arranged set of features [19]. This feature model handles the variability of both, the CPS and the test system. From this feature model, the optimal configurations are set in order to satisfy the maximum coverage. In addition, a Simulink model is generated automatically integrating the configured CPS and the test architecture.
- A configurable test architecture in Simulink that handles variability issues for the validation of configurable CPSs. This test architecture, based on [16], will include test data generators and test oracles handling variability as well as other components. The variability of the hardware and the software architecture requires variability in the test model at the verification and validation stages [7]. The test architecture together with the simulated CPS will get configured according to the chosen configuration.
- Algorithms for the efficient validation of configurable CPSs achieving high test coverage will be studied. The algorithms will combine dynamic CIT techniques, test suite minimization approaches and test case prioritization strategies. These algorithms will choose the most optimal configurations and will select and prioritize test cases to be executed for the chosen configuration.
- Test quality metrics for highly configurable CPSs. As mentioned before, it is infeasible to test all the possible configurations in highly configurable systems, and as a consequence, the notion of the achieved test coverage is uncertain. Different kinds of test quality metrics will be analysed to ensure the correctness and quality of the highly configurable CPS for any configuration.

V. Results

In our previous work [25], some experiments have been performed for the automatic generation of the CPSUT, where a novel variability modelling methodology is proposed for the plant models of highly configurable CPSs. In these experiments, we use FeatureIDE [18], a feature modelling tool for managing variability. With this tool we manage the variability of the physical side of the CPSUT and the .xml file is generated. With the .xml file, a first Simulink model is generated semi-automatically. Human intervention is needed to integrate the different components (sensors, actuators, mechanical components, embedded systems) of the model. Once generated this model, configurations are achieved either manually or automatically from Feature IDE, and each configuration returns a ".config" file. With this file, the configurator configures the model of the CPSUT.

The expected results include a reduction on the time needed for the validation of highly configurable system, at the same time as incrementing the obtained test coverage (requirements coverage, feature coverage, components coverage, etc.). This goal can be achieved by reducing simulation time through the analysis of the most effective variability modelling methodology. In addition, it is essential to select the optimal CPS configurations to be tested and to automate the simulation framework. In regard to test cases, it will be important to select the appropriate ones, prioritizing them and minimizing the test suite as much as possible.

VI. Conclusion and Future Work

This paper identified the main challenges to face when validating highly configurable systems. The main problems can be summarized into (1) the automatic configuration of the CPS and the test infrastructure, (2) the unclear notion of the achieved coverage and (3) the need for reducing test execution time. As a possible solution, we propose a model-based testing methodology to efficiently and systematically validate highly configurable CPSs.

A highly configurable CPS can achieve different configurations, in the software as well as in the hardware, which leads to the need of automating the configuration of the model of the CPSUT as well as the configuration of the test system at verification and validation stages. The described methodology proposes the automatic generation and configuration of the test system and the CPSUT for Simulink models from Feature Models.

A highly configurable CPS can be configured into thousands or even millions of configurations. Testing each possible configuration is impracticable. Hence, the notion of the achieved test coverage is uncertain. Optimizing the execution time of test cases as well as configurations to be set up is essential. We propose dynamic multi-objective algorithms with the aim of achieving the highest possible test coverage (requirements, feature and component coverage), using the minimum CPS configurations and the minimal test execution time.

VII. Acknowledgements

This work has been developed by the embedded systems group supported by the Department of Education, Universities and Research of the Basque Government.

References

[1] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," Proceedings of the IEEE (special issue on CPS), vol. 100, no. 1, January 2011, pp. 13 – 28.

[2] T. Yue, S. Ali, and K. Nie, "Towards a search-based interactive configuration of cyber physical system product lines," in ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems, Poster, 2013, pp. 71–75.

[3] T. Bauer et al., "Combining combinatorial and model-based test approaches for highly configurable safety-critical systems," in 2nd Workshop on Model-based Testing in Practice, 2009.

[4] J. V. Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in Proceedings of the Working IEEE/IFIP Conference on Software Architecture, ser. WICSA '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 45–54.

[5] S. Thiel and A. Hein, "Modelling and using product line variability in automotive systems," IEEE Software, vol. 19, no. 4, 2002, pp. 66 – 72.

[6] R. Behjati, T. Yue, L. Briand, and B. Selic, "Simpl: A product-line modeling methodology for families of integrated control systems," Simula Research Laboratory, Tech. Rep., 2011.

[7] D. Streitferdt et al., "Model-based testing of highly configurable embedded systems in the automation domain," International Journal of Embedded and Real-Time Communication Systems, 2011, pp. 22–41.

[8] C. H. P. Kim et al., "Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 257–267.

[9] S. Weißleder and H. Lackner, "Top-down and bottom-up approach for model-based testing of product lines," in MBT, 2013, pp. 82–94.

[10] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, "Combinatorial software testing," IEEE Computer Society, vol. 42, 2009, pp. 94–96.

[11] S. Oster, I. Zorcic, F. Markert, and M. Lochau, "Moso-polite - tool support for pairwise and model-based software product line testing," in VaMoS, 2011, pp. 79–82.

[12] S. Wang, S. Ali, and A. Gotlieb, "Minimizing test suites in software product lines using weight-based genetic algorithms," in Proceedings of the 2013 Genetic and Evolutionary Computation Conference, Amsterdam, Netherlands, 2013, pp. 1493 – 1500.

[13] G. Sagardui, L. Etxeberria, and J. A. Agirre, "Variability management in testing architectures for embedded control systems," in VALID 2012 - 4th International Conference on Advances in System Testing and Validation Lifecycle, Lisbon, Portugal, 2012, pp. 73 – 78.

[14] A. Arrieta, I. Agirre, and A. Alberdi, "Testing architecture with variability management in embedded distributed systems," in IV Jornadas de Computación Empotrada, ser. JCE 2013, 2013, pp. 12–19.

[15] B. Magro, J. Garbajosa, and J. Perez, "A software product line definition for validation environments," in 12th International Software Product Line Conference (SPLC), Piscataway, NJ, USA, 2008, pp. 45 – 54.

[16] J. Zander-Nowicka, "Model-based testing of real-time embedded systems in the automotive domain," Ph.D. dissertation, Technical University Berlin, 2008.

[17] T. Berger et al., "A survey of variability modeling in industrial practice," in Variability Modelling of Software-intensive Systems (VaMoS), 2013, pp. 7:1–7:8.

[18] T. Thuem et al., "Featureide: An extensible framework for feature-oriented software development," Science of Computer Programming, vol. 79, 2014, pp. 70 – 85.

[19] D. Benavides, S. Segura, and A. Ruiz-Corts, "Automated analysis of feature models 20 years later: A literature review," Information Systems, vol. 35, no. 6, 2010, pp. 615 – 636.

[20] A. Haber et al., "First-class variability modeling in matlab/simulink," in ACM International Conference Proceeding Series, 2013, pp. 4:1–4:8.

[21] G. Botterweck, A. Polzer, and S. Kowalewski, "Using higher-order transformations to derive variability mechanism for embedded systems," in Models in Software Engineering. Workshops and Symposia at MODELS 2009, Berlin, Germany, 2009, pp. 68 – 82.

[22] C. Dziobek, J. Loew, W. Przystas, and J. Weiland, "Functional variants handling in simulink models," Mathworks, Tech. Rep., 2008.

[23] M. Z. Z. Iqbal, "Environment model-based system testing of real-time embedded systems," Ph.D. dissertation, University of Oslo, 2012.

[24] H. Shokry and M. Hinchey, "Model-based verification of embedded software," Computer, vol. 42, no. 4, 2009, pp. 53 – 59.

[25] A. Arrieta, G. Sagardui, and L. Etxeberria, "Towards the automatic generation and management of plant models for the validation of highly configurable cyber-physical systems," in Proceedings of 2014 IEEE 19th Conference on Emerging Technologies & Factory Automation, ser. ETFA 2014, no. (To be published), 2014.

Aspect-Oriented Testing of a Rehabilitation System

Külli Sarna and Jüri Vain
Department of Computer Science
Tallinn University of Technology
Tallinn, Estonia
Kylli.Sarna@eliko.ee; Juri.Vain@ttu.ee

***Abstract*— The paper focuses on modularizing test models by adapting aspect-oriented modelling techniques. Model-based testing is an unavoidable part of contemporary model-driven software processes. The essence of model-based testing is to provide methods and tools to validate software systems by generating test cases systematically from models. From the practical usage point of view, it is critical to construct models that capture the essential aspects of the system under test. The proposed test design approach allows systematic separation of testing concerns, that, in turn, helps to overcome the complexity issues. Also, verification conditions are proposed to ensure the correctness of derived aspect test models and their compatibility with base test models. We demonstrate the technique of test model construction using timed automata models and illustrate it with a home rehabilitation system case study.**

Keywords-aspect-oriented testing; model-based testing; test model design; test generation.

I. INTRODUCTION

In the current practice of software testing, including Model-Based Testing (MBT), the test cases are frequently insufficiently structured and specified. The test designers use component-based or hierarchical state models. However, these modelling approaches provide poor support for isolating crosscutting features, specifically, functions that are spread across the software modules and tangled with other functions. We use the principles of Aspect-Oriented Modelling (AOM) to modularize such crosscutting functions into aspects. The AOM approach has evolved from aspect-oriented programming [2] to produce well-structured and well-encapsulated software. We enhance MBT design methodology with aspect handling capabilities taken from AOM [3]. Using the principles of AOM we can encapsulate typical cases like specifying requirements (use cases) that do not specify one property (scattering) or different functionalities (tangling). In this paper, we will explain how to conceptualize concerns into aspects and how to extract test cases from these aspect test models.

In MBT, the tests are generated from formal models of the System Under Test (SUT). The AOM technique introduced by Sarna and Vain [9] models SUT using timed automata and defines aspect models as refinements of the base model. The structural test coverage criteria considered are the same as those commonly used in state models, i.e., state, and transition coverage. As a novelty, in this paper we demonstrate how a test suite can be generated according to structural units that are specific to AOM. This gives us new test coverage criteria that address implemented features – aspect, advice, join-points coverage, etc. - and provide more intuitive reference to the parts of SUT to be tested for those features.

Another advantage of Aspect-Oriented (AO) MBT is the possibility of easy modification of the test suite. When new requirements arise, new advice models can be woven into the test suite without redesigning the existing base model.

Applying the principles of AOM does not provide compositional testing techniques *per se*. Compositionality of proposed AO testing is achieved by imposing extra constraints on how the advice models are constructed and model weaving operations defined. We define these rules in the semantic framework of Uppaal timed automata [6] and formulate the proof obligations to be model-checked. Our approach is illustrated with a home rehabilitation system testing framework.

The rest of the paper is structured as follows. We introduce the technical background in Section 2. Section 3 describes AO MBT. In Section 4, the home rehabilitation system is introduced. Finally, Section 5 concludes the paper.

II. BACKGROUND

A. Aspect-oriented modelling

AOM is a way of modularizing *crosscutting concerns* much like object–oriented programming is a way of modularizing *common concerns*. *Crosscutting concerns* generally refer to non-functional properties of software, such as security, synchronization, mobility, resilience, etc. In addition, every system may contain its own application specific crosscutting concerns [5].

Cottenier et al. [4] and Rashid [8] have admitted that AOM technologies have the potential to simplify software deployment, and the ability to improve the categorization of crosscutting concerns. Also, AOM aids in modular extension of object systems, where the treatment of crosscutting concerns is encapsulated in separate modules called *aspects*. We use concepts taken from AOM, such as *Aspect, Advice, Join-points, Pointcut*, and *Weaving.*

An aspect consists of two parts: the code/model associated with treatment of the concern (called *advice*), and a predicate defining when the advice should be applied during system executions (called a *pointcut*). The points in the code/model that are identified by a pointcut are called *join-points*.

A *pointcut* selects a subset of join-points based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards. *Pointcuts* can be composed using logical operators. Customized *pointcuts* can be defined, and *pointcuts* can identify *join-points* from different aspects. The process of adding aspects to a base system is called *weaving*; and the result is referred to as the *woven system* [5]. AOM techniques use the term *advice* for the action an aspect will take and *join-points* for where these actions will be inserted in the base system model. *Pointcuts* are used to specify the rules of where to apply an aspect. *Advice*, *join-points*, and *pointcut* are specified as one entity, called an *aspect* [7].

As in AOM, AO testing uses a *base test model* and several *aspect test models.* An example of a base test model is depicted in Figure 1 (for better understanding of the relationship between the models, we use an Automatic Teller Machine (ATM) as an example of a well-known system). The ATM test model specifies the use case of withdrawing money from an ATM. Crosscutting features are treated as patterns described by aspect advice models, and common features are described in the base model. The result of weaving the base model with advice models is called the *composed aspect model*. An advice model can be woven with the base model in many places and in different ways. The Transaction advice model is defined as location refinement of both ATM and Customer automata. The details of advice model construction in the test design level are presented in [9].

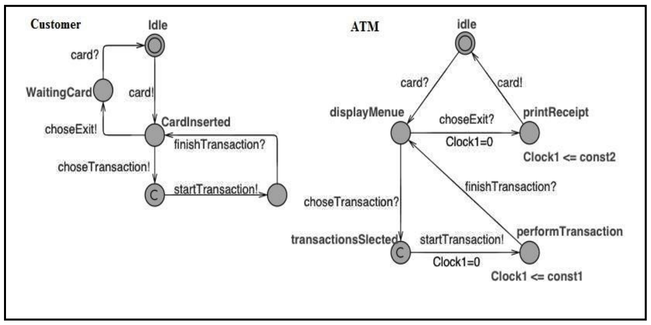


Figure 1. The base test model of ATM.

The base model of an ATM depicted in Figure 1 includes interacting Customer and ATM automata. Refinements in Figure 2 specify aspects of interest: (i) the Transaction advice model is defined as location refinement of both ATM and Customer automata; (ii) edge refinement of ATM. The aspect behaviour is launched from the base model explicitly with the help of channels. We model in Uppaal (www.uppaal.com), a tool box for modelling, simulation and verification of timed automata. In Uppaal [12], the synchronization mechanism is a hand-shaking synchronization: two processes go through a transition at the same time, one will be labelled x !, and the other x ?, where suffixes ?, and ! after the channel name x distinguish sending and receiving synchronization information respectively. A system is composed of concurrent processes, each of them modelled as an automaton. The automaton has a set of locations and edges to specify the control flow. A transition specified by an edge is enabled if its guard and synchronization conditions are satisfied. The transaction automaton in Figure 2 introduces the EnquireBalance aspect

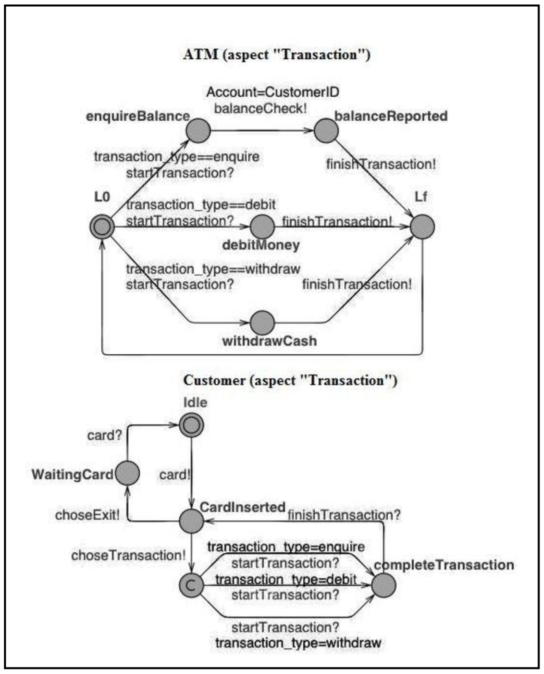


Figure 2. The aspect model “Transaction”.

advice. Since the refinement (ii) introduces a new interaction between ATM and a new actor Server (not shown in the model) the edge introduced is labelled with the ‘balanceCheck!’ channel. When the aspect related tests have to be generated from the composed model of SUT that includes the automata in Figures 1 and 2, we can ignore all the transactions that the aspects of interest do not depend on. For instance, when testing the balanceCheck! transaction between ATM and Server the tester model is extracted from the composition Customer || Customer(Transaction) by algorithm of [1] so that the test sequence <card!, choseTransaction!, transaction_type := enquire, start-Transaction!, wait,[finishTransaction?| timeout >= const1, TESTFAIL], choseExit!, card?, TESTPASS > can be executed.

B. Model-based testing

MBT uses abstract behavioural models for specifying the expected behaviour of the SUT and for automatically generating tests to check if the behaviour of SUT conforms to the model. The SUT is an executable implementation which is considered as a black-box during the testing process, i.e., only inputs and outputs of the system are visible externally. The SUT is tested incrementally by applying test cases. A test case in MBT is defined as a sequence of test stimuli paired with expected SUT outputs. A specified set of test cases constitutes a test suite.

C. Uppaal timed automata

Assume a finite alphabet Σ ranged over by *a*, *b*,... stands for actions and a finite set *C* of real-valued variables ranging over by *x*, *y*, *z*, standing for clocks.

A guard is a conjunctive formula of atomic constraints of the form $x \sim n$ for $c \in C$, $\sim \in \{\geq, \leq, =, >, <\}$ and $n \in$ N. We use $G(C)$ to denote the set of guards, ranged over by g.

Definition 1 (**Timed Automaton**) [6]
A timed automaton $\mathbf{A}$ *is a tuple* $\langle N, l_0, E, I\rangle$ where
– N is a finite set of locations (or nodes),
– $l_0 \in N$ is the initial location,
– $E \in N \times G(C) \times \Sigma \times 2^C \times N$ is the set of edges and
– I: $N \to G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $x \leq n$ or $x < n$, $n \in$N. For shorthand we write $l \to_{g,a,r} l'$ to denote $\langle l, g, a, r, l'\rangle \in E$.
To model concurrent systems, timed automata are extended with parallel composition. In the UPPAAL modelling language, the CCS parallel composition operator is used, which allows interleaving of actions as well as hand-shake synchronization. The parallel composition of a set of automata is the product of the automata.
The semantics of timed automata is defined as a transition system where configuration consists of the current location, valuation of state variables and the current values of clocks.
There are two types of transitions between states: the automata may either delay for some time (delay transition), or follow an enabled edge (action transition).

To keep track of the changes of clock values, we use functions known as clock assignments mapping C to the non-negative reals R$_+$. Let u, v denote such functions, and $u \in g$ means that clock values denoted by u satisfy the guard g. For $d \in$ R$_+$ let $u + d$ denote the clock assignment that maps all $x \in C$ to $u(x) + d$ and for $r \subseteq C$ let $[r \mapsto 0]$ denote the clock assignment mapping all clocks to 0 and agree with for the other clocks in $C\backslash r$.

Definition 2 (**Operational Semantics**) [6]
The semantics of a timed automaton is a transition system (also known as a timed transition system) where states are pairs $\langle l, u\rangle$ and transitions are defined by the rules:
– $\langle l, u\rangle \to_d \langle l, u + d\rangle$ if $u \in I(l)$ and $(u + d) \in I(l)$ for a non-negative real $d \in$ R$_+$
- $\langle l, u\rangle \to_a \langle l', u'\rangle$ if $l \to_{g,a,r} l'$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$.
To increase the modeling power keeping the analysis traceable for planner synthesis we lift the model class to rectangular timed automata where guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints of integer variables.

Similarly to clock conditions, the integer variable conditions are of the form $k \sim n$ for $k \in$ Z, $\sim \in \{\geq, \leq, =, >, <\}$ and $n \in$ N. The advantage of this extension is that the model has rich enough modelling power to represent real-time and resource constraints being same time efficiently decidable for reachability analysis.

III. Aspect-Oriented Model-Based Testing

In this section, we explain the concepts of AOM applicable in aspect-oriented MBT. The AOM allows the models to be organized so that they address particular requirements (including crosscutting ones) and corresponding test cases. The AO test model includes a base model and aspect-related advice models. Aspects may contain sub-aspects that require sub-advices and their own test cases. Sub-aspect models have to be easily inserted into their parent aspect models. In our examples, we use name prefixes that refer to the parent models so that they are convenient to comprehend and maintain.

AO testing can also be considered as an example of compositional testing where the test results of the composed system can be inferred from the test results of its components. In the MBT context, it means that the test cases are determined only by the context of the aspect advice models and the interface behaviour of their composition. AOM also provides a conceptual basis for defining test coverage criteria in terms of aspect related model elements. The hierarchy of those criteria is depicted in Table I.

TABLE I. AO TEST COVERAGE CRITERIA

Coverage constraint / Type of coverage entity	Strong (universal) coverage $\forall$	Weak (existential) coverage $\exists$	Discriminating predicate
Aspect A	All aspects of the model $\forall A \in \mathbf{A}.$ ...	Some aspects of the model $\exists A \in \mathbf{A}.$ ...	Predicate on aspect constants /variables
i-th join point $jp(A, i)$	All join points of aspect A $\forall jp(A, i) \in JP(A)$	Some join points of aspect A $\exists jp(A, i) \in JP(A)$	Point cut condition
Entry-exit path λ of an advice model $M^{A'}$ $\lambda \in Paths(M^{A'})$	All paths initiated at i-th join point $\forall \lambda \in Paths(M^{A'})$	Some paths initiated at i-th join point $\exists \lambda \in Paths(M^{A'})$	Path predicate, e.g. constraint on path length
Model element of type $\boldsymbol{T}$ (location, transition, function, data, etc) included in the path $\lambda \in Paths(M^{A'})$	All elements of type $\boldsymbol{T}$ in $M^{A'}$	Some elements of type $\boldsymbol{T}$ in $M^{A'}$	Predicate on the attributes of type $\boldsymbol{T}$

The criteria shown in Table I can be expressed as closed 1st order logic formula in prenex normal form, where the signature includes variables of particular types of structural elements of Uppaal Timed Automata (UPTA) (template, location, transition, label, function, data, etc.). The prefix of the prenex formula includes bound variables in a fixed order that is determined by the natural hierarchy of modelling entities: aspect, join-points, and path. These entities model the structural elements of UPTA, where the structural elements can be referred to directly by name or indirectly by constraints on their attributes. The matrix part may include discriminating predicates of all the above listed types.

The semantics and scoping of AO coverage constraints is defined by the hierarchy and type structure of AO model elements (left most column in Table I). Thus, the scope of constraints on bound variable in the formula matrix part is defined by the position of the bound variable in prefix. For instance, the scope of a path constraint is defined by the join-point and aspect constraints because these elements precede path variable in the prefix. When not explicitly expressed in coverage constraint the default scoping means existential quantification over all those variables preceding in the prefix of coverage constraint. For characterization of coverage criteria in terms of Uppaal query language, we assume that the aspect model *M* is constructed according to the rules described in [9]. The idea is to use Uppaal model checker queries for selecting traces that constitute the test paths of the given test case. Uppaal query based online test generation methods are described by Vain et al. [1] and Hessel et al. [10].

Aspect Coverage criteria impose to execute all or some aspects in a woven model at least once. In *Strong Aspect Coverage* (SAC), given an aspect model *M*, all possible test paths must be covered by the tests. To implement the Strong Aspect Coverage we use the parameterized UPTA templates where the template parameter p_i ranges over indexes [1, *n*] that identify the aspect. Let `P(i)` be a predicate updated to true whenever the i-th aspect advice model is entered. Then the traces of *M* (p_i) under Strong Aspect Coverage criteria should satisfy the query: `E<> forall (i: int [1,n]) P(i)`. Note that given query is valid only for paths that include traversal of all aspects' advice models. In general, the model *M* may not be fully connected and a single path including all aspects may not exist. Therefore, we introduce an auxiliary *reset*- transition into *M* that guarantees that if *n* advice models are reachable in *M* then at most with *n* traversals all of them are visited. The *reset*-transition connects the final location of *M* with its initial location. Due to this construct the Uppaal model checker is able to generate a trace that includes visits of all advice models. The tests paths for a final test case can achieved simply by "cutting" that trace at *reset*- transitions to many shorter sub traces.
Weak Aspect Coverage (WAC) refers to the case where at least one advice model of some aspect is traversed by the test path. The query `E<> forall (i:int [1,n]) P(i)` differs little from the strong coverage constraint but it does not require including *reset*-transitions in the model *M*.

Join Point Coverage criteria impose to execute all or some join points of each aspect in a woven model at least once. *Strong Join Point Coverage* (SJPC) presumes similarly to strong aspect coverage introduction of an auxiliary *reset*-transition into *M*. Regardless the prefix (SAC or WAC) of the query the SJPC contributes a conjunct of form ...`forall (j: int [1,m]) P(i) && R(j)` where `j` is ranging over join point indexes of the aspects referred in the prefix of that query and `R(j)` is a Boolean variable at each join point updated to `true,` whenever this join point is visited. *Weak Join Point Coverage* (WJPC) is satisfied if there is at least one trace for given formula prefix satisfying ...`exists (j: int [1,m]) P(i) && R(j)`. Here, like in WAC, auxiliary *reset*-transition is not needed.

Aspect Path Coverage criteria impose to execute all or some paths of each aspect in a woven model at least once. Assume the entry and exit transitions of each advice models are decorated with *entry*(*i*, *j*,*k*) and *exit*(*i*, *j*,*l*) predicates where *i*, *j*, *k*, *l* range over the set of aspects, join points, and their advice entry and exit points respectively. Whenever the transition is executed these predicates evaluate to `true`. Then, the *Strong Aspect Path Coverage* (SAPC) contributes a conjunct to the query prefixed with aspect and join point constraints as follows: ... `forall (k: int [1,K]) forall (l: int [1,L]) P(i) && R(j) && [(`$\vee_{k=1,K}$ `entry(k))` $\wedge$`(`$\vee_{l=1,L}$ `exit(l))`. SAPC, like earlier strong coverage criteria, presumes the *reset*-transitions related construct. *Weak Aspect Path Coverage* (WAPC) comparing to SAPC replaces universal quantifiers with existential ones for variables k and l, the coverage constraint becoming to ... `exists(k: int[1,K]) exists(l: int[1,L]) P(i) && R(j) && [(`$\vee_{k=1,K}$ `entry(k))` $\wedge$ `(`$\vee_{l=1,L}$ `exit(l))`.

The **Model Element Coverage** criteria impose constraints on the types of UPTA elements to be covered in the advice model or set the specific constraints on the attributes of those elements, e.g. *Strong* (resp. *Weak*) *Model Element Coverage* can be parameterized with the element type, e.g. `Transition` and universally (resp. existentially) quantified over given type. More specific coverage constraints can be constructed using type discriminating predicates on, e.g., local data variables of an advice model.

IV. EXAMPLE: TESTING HOME REHABILITATION SYSTEM

The AO MBT approach described in Section 3 has been applied in testing a Home Rehabilitation System (HRS). The model-based testing is needed in the medical domain because of the safety critical nature of the systems and non-trivial combination of functional, performance and security features [11]. The HRS is an application which drives sensor devices, analyses the gathered data, interacts with the patient and submits relevant information to the hospital through the Internet. HRS software contains the following subcomponents: dedicated health hub as communication gateway; vital signals' sensor system for patient measurements; movement tracking sensor system for fall detection, physical activity and exercise monitoring.

There are three actors, namely, Patient, Plan and Sample, interacting in the "home exercising" use case. The composition of automata *Plan* and *Sample* constitute the base model that can be woven with different advice models depending on what body characteristic (pulse, blood pressure, etc.) is monitored. For instance in Figure 3, the use-case *exercising* is refined with two advice models that are instances of the same automaton template. The advice models linked to the base model are location refinements of the unnamed location in the automaton *Sample*. Channel *Sample* ensures that the advice models are executed synchronously with the edge departing from location *Measure* in the automaton *Sample*. A weak join point coverage of completing exercising can be specified now using query `E<>exists(Screen=UB_warning[1])`. The test case ensures that while a patient is exercising, a warning will be shown on a screen when the patient's pulse is greater than the number in U_bound. On the other hand U_bound is the upper value of pulse that the patient may have during exercising and this is specific to each patient. For example if the U_bound is 140 then a warning on a screen goes red and warns "wait until your pulse will be normal". We measure the pulse under "measurement [1]" and an upper bound and a lower bound are indicated. A normal pulse measurement have to be between U_bound and L_bound.

A strong join point coverage of completing exercising can be specified using query `E<>forall (Screen=normal[1])measurement[1]>=L_bound [1]&&measurement[1]<=U_bound[1]`. That means the screen indicates in green that everything is alright and the patient can continue exercising because their pulse is within the allowed range. By this strong join point test coverage, we ensure that our system is able to give the right warnings whenever necessary.

V. CONCLUSION

In this work, we have introduced an aspect-oriented approach to model-based testing in the context of Uppaal timed automata specifications. We advocate the view that aspect-oriented models help in constructing models of system under test in a systematic and user friendly way, thus helping to defeat the perennial problems of MBT - complexity of construction and maintenance of test models. It has been shown how the aspect related test coverage criteria can be formalized in a systematic way in Uppaal query language Timed Computation Tree Logic (TCTL) and the feasibility of test suites verified on aspect models before real tests are deployed and executed.

Our focus on how a test case can be generated according to structural units that are specific to AOM is novel. This gives new test coverage criteria that address implemented features – aspect, advice, join-points, etc., and provide more intuitive reference to the parts of SUT to be tested for those features.

Another contribution for enhancing MBT by aspects is the possibility of easy update of test case related models. If new requirements arise, new advice models can simply be incorporated by well-defined composition rules. This is especially relevant in regression testing.

ACKNOWLEDGMENT

This research was supported by the European Social Fund's Doctoral Studies and Internationalisation Programme (DoRa), and by the Competence Centre Programme of Enterprise Estonia.

REFERENCES

[1] J. Vain, M. Kääramees, and M. Markvardt, "Online testing of nondeterministic systems with reactive planning tester," in: L. Petre, K. Sere, and E. Troubtsyna (Eds.). Dependability and Computer Engineering: Concepts for Software-Intensive Systems. Hershey, PA: IGI Global (2012), pp. 113-150.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. - M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", ECOOP'97, June, 1997, pp. 220-242.

[3] J. Kienzle, A. Wisam, F. Fleury, J. Jezequel, and J. Klein, "Aspect-Oriented Design with Reusable Aspect Models." Transactions on Aspect-Oriented Software Development, vol7, 2010, pp. 279-327.

[4] T. Cottenier, A. van den Berg, and T. Elrad, "Stateful Aspects: The Case for Aspect-Oriented Modeling." Proceedings of the 10th AOM Workshop, 2007, pp. 7-14.

[5] E. Katz, and S. Katz, "User queries for specification refinement treating shared aspect join points." Proceedings of the 8th IEEE, 2010, pp. 73-82.

[6] J. Bengtsson, and W. Yi, "Timed automata: Semantics, algorithms and tools." Lecture Notes on Concurrency and Petri Nets, Lecture Notes in Computer Science vol. 3098, 2004, pp. 87-124.

[7] S. Clarke, and E. Baniassad, Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley Professional. 2005.

[8] A. Rashid, "Aspect-Oriented Requirements Engineering: An Introduction". In Proceedings of 16th IEEE, 2008, pp. 173-182.

[9] K. Sarna, and J. Vain, "Exploiting Aspects in Model-Based Testing," in Proceedings of 11th FOAL. ACM, New York, NY, USA, 2012, pp. 45-48.

[10] A. Hessel, K. G. Larsen, P. Pettersson, and A. Skou," Testing Real-Time Systems Using UPPAAL." Lecture Notes in Computer Science vol. 4949. 2008, pp. 77-117.

[11] A. Kuusik, E. Reilent, K. Sarna, and M. Parve, "Home telecare and rehabilitation system with aspect-oriented functional integration." Biomedical Engineering, DOI: 10.1515/bmt-2012-4194 (accessed 01.08.2014).

[12] K. Larsen, P. Pettersson, and W.Yi. UPPAAL in a nutshell. Journal on Software Tools for Technology Transfer, 1997, pp. 134-152.

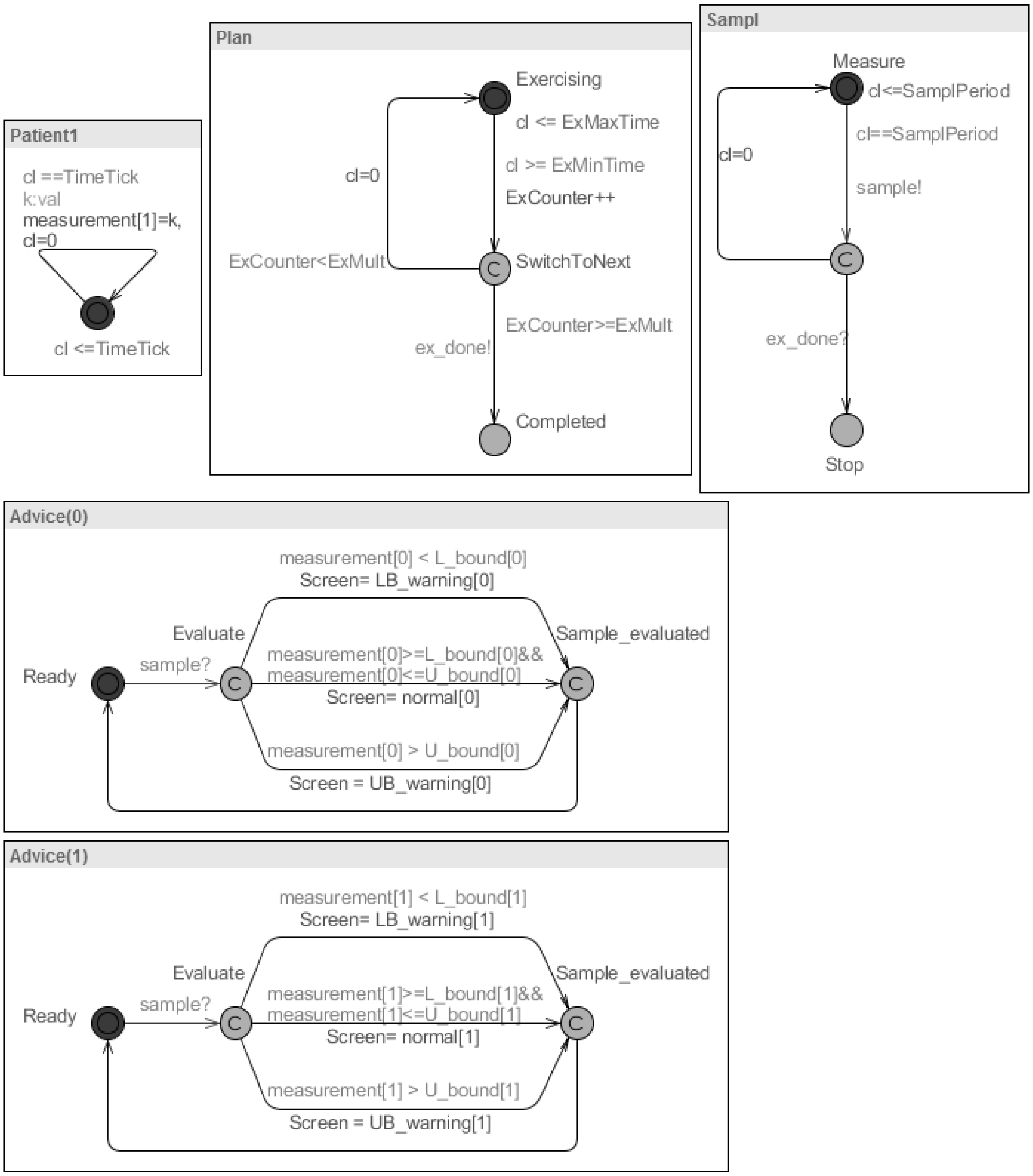


Figure 3. Composing the primary test models and advice model in parallel.

A Configurable Test Architecture for the Automatic Validation of Variability-Intensive Cyber-Physical Systems

Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria

Computer and Electronics Department
Mondragon Goi Eskola Politeknikoa
Email: {aarrieta, gsagardui, letxeberria}@mondragon.edu

***Abstract*—The strong demand for customizable products is leading to increase variability in cyber-physical systems. The need of dealing with variability issues increases complexity not only in the product, but also in the verification and validation activities. Due to the high amount of configurations that the system can be set to, verification and validation activities might become time consuming and non-systematic. In order to deal with these problems, this paper presents an automatically configurable test architecture together with a model-based process for the systematic validation of highly configurable cyber-physical systems, with the main objective of reducing verification and validation costs. The main contributions of this paper are the analysis of the variability of the test system and its components together with the traceability among the features of the cyber-physical system and the test system and a definition of a model-based process to achieve the test objectives.**

Keywords–Model Based Testing; Test Architecture; Variability; Configurable Systems.

I. Introduction

Modern society is depending on Cyber-Physical Systems (CPSs) in charge of controlling many individual systems and complicated coordination of those systems [1]. CPSs combine digital cyber technologies with physical processes, where real-time embedded and networked systems monitor and control physical processes with sensors and actuators [2]. Industrial CPSs are highly complex systems [1], and variability increases in order to deal with the changes that the physical environment requires.

Variability is defined as the ability to change or customize a system [3], which can be understood as configurability (variability in the product space) or modifiability (variation or evolution over the time) [4]. Variability in CPSs can appear as configurability when different components or functionalities can be chosen or are optional depending on the customer's needs or budget, e.g., different temperature sensors with different precision. In addition, modifiability appears when a new feature or functionality is added to the CPS, e.g., apart from measuring temperature, humidity is wanted to be measured. Run-time variability is also common in CPSs, which permits the adaptation to changes in the environment.

Testing configurable CPSs can become a very time and resource consuming activity. This is, to a large extent, caused by the high number of possible variants, which give to the system the chance of being set into thousands of configurations, making the testing of all the existing combinations really infeasible. The high number of variants and their complexity make manual configuration of variability-intensive CPSs error-prone and inefficient, which warrants the need for an automated solution for CPS configuration [1], as well as for its test system.

Traditional software testing activities can reach even the 50 % of the development costs [5], which can be incremented if the System Under Test (SUT) is highly configurable. Model-Based Design (MBD) tools help in the development of embedded software, a task which is becoming more and more complex, especially when variability issues have to be taken into account. Model-, Software-, Processor- and Hardware-in-the-Loop (MiL, SiL, PiL and HiL) tests, provide four testing phases [6], which are typically used to test CPSs in different stages and testing objectives.

According to Berger et al. [7], the automotive domain is the industry where variability modelling is most used, and MATLAB/Simulink is a modelling language widely used for modelling embedded software in this domain [8], as well as for simulating CPSs. This has been the main reason that has motivated us to choose MATLAB/Simulink as simulation framework to achieve our objective.

Test architectures are the organization of the group of components in charge of testing a system. A test architecture is a necessary artefact for a testing process [9], so that verification and validation activities can be systematic, allowing among other advantages reusability of test cases along the different test phases. According to Nishi [10], test architectures can be differentiated into test system architecture and test suite architecture. The test system architecture focuses on issues such as test system, SUT, platform where the SUT is executed or test case generation, whereas the test suite architecture relies on groups of test cases, test levels, etc. [10]. The viewpoint of this paper will focus on the test system architecture.

Two main challenges have been identified to efficiently test highly configurable CPSs. On the one hand, the definition of a test architecture that automatically gets configured for the selected configuration of the CPS. Each configuration of the CPS is different, with different ports, parameters, functionalities and requirements; therefore, the test architecture must be adapted to the selected configuration. On the other hand, the selection of the test cases that each configuration must execute. Each configuration requires a set of test cases to be executed depending on the selected components and the product requirements. In addition, it might be possible that some test cases had already been executed in a similar configuration and that can be avoided to save time.

This paper focuses on the first issue, how to automatise the configuration of a test architecture for configurable CPSs, which serves as a basis to develop a configurable test architecture in Simulink. Moreover, this test architecture is able of getting configured automatically taking into account the selected configuration for the configurable CPS. In addition, we propose a systematic process that enables reducing the validation time of configurable cyber-physical systems considerably.

The rest of this paper is structured as follows. Section II introduces the related work. Section III presents the configurable test architecture and the components composing it. In Section IV, the necessary steps to systematically test variability-intensive CPSs are introduced. Preliminary as well as expected results are described in Section V. Finally, Section VI provides the conclusions of this paper.

II. Related Work

Model-in-the-Loop for Embedded System Test (MiLEST) is a toolbox for MATLAB/Simulink developed by Zander-Nowicka [11]. The test architecture, depicted in Figure 1, is based on MiLEST, specially the test oracle and the test data generator. The hierarchy of MiLEST is divided into four abstraction levels: Test Harness level, Test Requirement level, Test Case level and Feature level. The main difference between the test architecture proposed in [11] and our test architecture is that the test architecture presented in this paper supports variability issues and we add an additional block named "Test Historic Database" with the main objective of reducing testing time.

From the testing perspective, our previous work [9] presents an approach based on model-based testing and variability management integrated in Simulink, where a concrete configuration of the software is chosen by the test engineer, and the testing infrastructure is instantiated for the chosen configuration; the main shortcomings of this approach are that the plant model is not simulated and that important components such as test oracles do not handle variability. In another previous work [12], a testing architecture with variability management in the test oracles is proposed to test distributed robotic systems in Simulink; in this case, the main limitations of this approach is that it was only oriented for same purpose distributed systems, and the variability points were only modelled in a test oracle.

A product line of validation environments with variability to test different applications in different domains and technologies is proposed by Magro et al. [13]. The main limitations of this approach are that the components handling variability are too high level components, it does not support test automation and variability is not managed with any tool.

Combinatorial testing is also widely used when testing configurable systems, where configurations are often selected using pairwise or t-wise techniques. Combinatorial testing techniques are commonly combined with model-based testing as proposed by Oster et al. [14], where a tool chain is introduced named MoSo-PoLiTe. This tool selects pairwise configuration selection component on the basis of a feature model covering 100% pairwise interaction, and test cases are generated for each configuration.

Nevertheless, this combinatorial technique selects product configurations statically. In [15], a novel approach is presented named SPLat, where product configurations are determined dynamically during test execution by monitoring accesses to configuration variables. The technique consists on executing the test for one configuration, while observing the values of configuration variables used to prune other configurations.

Another approach to reduce validation costs of highly configurable systems is minimizing the test suite for testing a product, reducing redundant test cases. A set of test cases can be automatically obtained by selecting features of a feature model to test a new product, but there still can exist redundant test cases [16]. A fitness function based on three key factors (Test Minimization, Feature Pairwise Coverage and Fault Detection Capability) for three different weight-based genetic algorithms is defined by Wang et al. [16]. This approach allows reducing the test suite covering all testing functionalities achieving a high fault detection capability.

III. The Configurable Test Architecture

A. *Variability in the Test Architecture*

As depicted in Figure 1, our test architecture is composed by five sources: System Under Test (SUT), Test Data Generator, Test Oracle, Test Control and Test historic Database. The experiments performed by Weißleder and Lackner [17] show that the most efficient way to test configurable systems is including variability down to the test case level. The test architecture must deal with variability so as to be configurable. The following list describes the different components and defines where variability can be found:

- System Under Test (SUT): it is the CPS to be tested. The components can be divided into hardware and software, where the software can be configured as model or software. In this approach, the SUT is kept as black-box, what means that the test engineer does not need to be familiar with its internal behaviour [18]. Variability in a CPS can be found in the cyber as well as in the physical side. Variability in the physical side (commonly known as context variability) is related to the variability of the environment. In a previous work [19], we proposed a methodology to manage and model variability in plant models for CPSs.
- Test Data Generator: it is the source in charge of stimulating the SUT with test signals. Variability in the test data generator can be found in signals (number and characteristic of each signal), requirements (number and parameters of requirements) and test cases (test case duration and test case characteristics).
- Test Oracle: it is the source where the SUT behaviour is examined and the test result is determined by a verdict [11]. Variability in the Test Oracle might be found in signals (number of inputs to the oracle) and requirements (number of requirements, number of validation functions, validation function characteristics and requirement parameters).

- Test Control: it is the specification that executes test cases [11]. In the case of systems with many variants, it is important to reduce the testing time as much as possible because there are many configurations that must be tested. This source will execute test suite minimization algorithms to reduce the overall validation time, as well as test case prioritization algorithms to reduce the time needed to detect faults. Once the tests are finished, the test controller will send information of the results to the test historic database.
- Test Historic Database: the results of the generated test cases are stored in a database in order to firstly, have information of the quality of the system and secondly, to avoid testing the same features many times. The test historic database would store the tested configurations, verdicts of test cases, achieved coverage, versions of the tested SUT, etc.

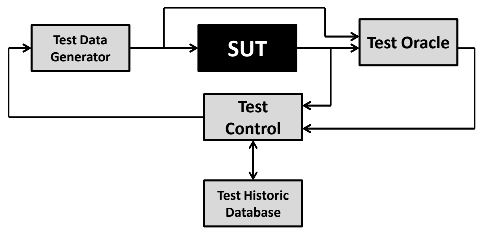


Figure 1. Configurable Test Architecture (based on [11]).

B. Managing and Modelling Variability

The previous section has analysed where the variability can appear in a test architecture. However, this variability must be managed and modelled. Feature Modelling is the most used technique in industry to manage and represent variability of systems [7]. A feature model can be defined as a notation that represents features and relations among them of all possible product configurations [20]. Basic feature models represent features as "mandatory", "optional", "alternative" and "or". Constraints among features also can appear in form of "Require" and "Exclude". Figure 2 depicts an example of a feature model.

MATLAB/Simulink can be used to model variability, as explained in several approaches in the literature. In [21], uniform variability is modelled using different mechanisms and blocks offered by MATLAB/Simulink, and variability is managed using the tool pure::variants [22]. In [8], different blocks are classified depending on their granularity, mapping of feature types, binding times and quality aspects. Another approach is presented Botterweck et al. [23], where variants are managed using a tool named S2T2 and variability is modelled in a similar way as explained in [21], but when the product is configured, the unselected features are erased from the model. Another interesting and different approach is presented in [24], named "Delta Simulink", where "Delta Modelling" is used in Simulink models to model variability. In our previous work [25], we compared the main characteristics of the variability modelling and management approaches documented in the current state of the art.

C. Traceability among Features, Feature Models and the Test Architecture Components

We believe that using feature models to manage variability of the testing infrastructure, as shown in Figure 2, may help to automatically generate the components that comprise the test architecture. The selected feature modelling tool is FeatureIDE [26], which is an open source plug-in for Eclipse that can be modified to adapt it to our needs.

FeatureIDE allows obtaining the .xml file of the developed feature model. Using a .xml parser in MATLAB it is possible to extract the data needed to automatically generate the test architecture.

FeatureIDE also permits an automatic configuration (all possible configurations, t-wise configuration, etc.) or a manual configuration of products, which returns a file with an extension ".config". This extension can be read from a model configurator developed in MATLAB that automatically configures the Simulink model (including the SUT and the test architecture) before running the test.

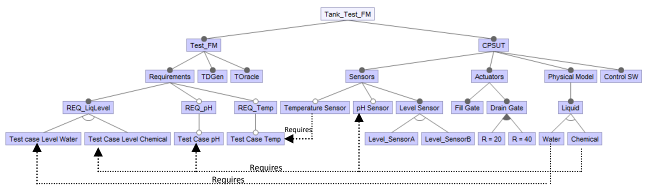


Figure 2. Example of how the Test Feature Model looks like.

We propose a motivating example involving the control of the liquid level of an industrial tank to better understand the proposed approach. The liquid can be water or a chemical product. When the liquid is a chemical product, it will be mandatory to measure its acidity with a pH sensor. Other variability points in the example are included: an optional temperature sensor, two different sensors to measure the liquid level and two types of gates to drain the liquid.

The test feature model would have several branches, which can be divided into branches for the test system (Requirements, TDGen, TOracle) and the branches for the CPSUT, and both are related. To achieve this, constraints ("requires" or "excludes") of the test feature model play an important role, as they define the relations among components of the product line and test cases. As shown in the example depicted in Figure 2, Temperature Sensor requires its corresponding test case, named "Test Case Temp". When chemical liquid is selected, the pH Sensor is also selected, and two test cases needs to be executed: "Test Case Level Chemical" and "Test Case pH". The same happens when the liquid is water.

IV. Systematic Model-Based Process for the Validation of Highly Configurable Cyber-Physical Systems

We consider that it is essential to follow a systematic process to test variability-intensive CPSs. There are many possible configurations, which makes impracticable to test all of them and there is uncertainty of the achieved test coverage. Figure 3 depicts the overview of the model-based process that enables the systematic validation of highly-configurable CPSs. The components of the process shown in Figure 3 are classified into two phases following the theory of product lines: Domain Engineering and Application Engineering. The main difference between these two phases is that the components in the domain engineering are for the whole product line, and therefore variability is implemented. On the contrary, the components corresponding to the application engineering are for a concrete product configuration, where variability is already bound. In addition, seven steps are needed to carry out the whole process:

- **Step 1:** The test feature model has to be developed. This step is one of the most important ones as it will allow to manage and handle the variability of the test system, and later this will be used to automatically generate the test architecture.
- **Step 2:** The .xml file of the developed test feature model is generated and saved to be manipulated by the test architecture generator. This .xml file is automatically generated by the tool FeatureIDE [26].
- **Step 3:** The test architecture generator, which is implemented in MATLAB, parses the .xml file of the feature model and saves the needed information of the test architecture and its variability in MATLAB cell arrays. With the saved information, the test architecture will automatically generate the Simulink model of the test system integrated with the Simulink model of the CPSUT using MATLAB scripts.
- **Step 4:** Configurations are generated either manually or automatically and stored in .config files in a library. These configurations are the ones that will later be tested. The tool FeatureIDE [26] allows either the manual or automatic generation of configurations. In the case of automatic configuration generation, combinatorial techniques, such as pairwise or t-wise, are used.
- **Step 5:** The test configurator, which is implemented in MATLAB, parses the configuration files and the components of the simulation framework are configured. The different possible test cases (handling variability), and the models of the hardware and software components will be stored in their corresponding libraries. The test configurator will allocate the needed components in the correct place of the simulation framework and integrate them, removing the components that are not needed for the configuration with the objective of saving simulation time.
- **Step 6:** When the simulation starts running, as mentioned before, the test controller will obtain information about previously executed test cases in order to prioritize and remove redundant test cases. Once processed the information about previously executed test cases, the simulation framework will test the selected configuration.
- **Step 7:** The executed test results are saved in the database, and ready to test another configuration. For the second test run, it will be enough to start from the fifth step.

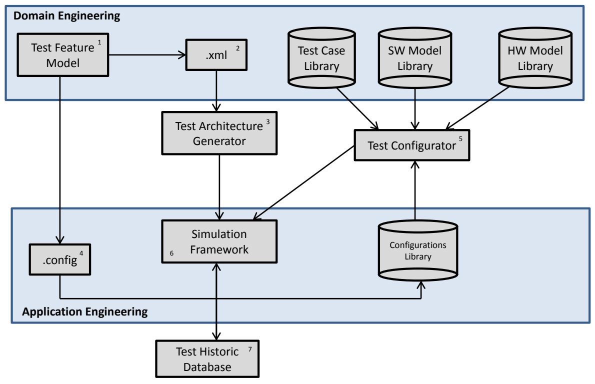


Figure 3. Overview of the Systematic Model-Based Process for the Validation of Highly Configurable Cyber-Physical Systems.

V. Results

Preliminary results include the semi-automatic generation and configuration of the SUT. In this process, we extract data from a feature model and generate automatically the components of the SUT's model in Simulink. The most challenging part is the automatic integration among the components, where the configurator needs information about the input and output ports of each component, i.e., their datatype, how are related with the other components, etc. The following step would be the automatic generation of the test architecture and its integration with the SUT.

The proposed test architecture in combination with the presented model-based process allows testing systematically real-time CPSs that have to deal with many variants. In addition, the use of a test controller, which is communicated with the test historic database, and that contains algorithms for different test objectives, as well as to prioritize test cases and to remove redundant test cases will considerably reduce verification and validation time achieving high test quality.

Another important issue for saving simulation time is the allocation of the components for the selected product, removing the features that are not specified in the configuration. We propose storing the different components in libraries and allocating them automatically in the Simulink model with the test configurator, as proposed by Arrieta et al. [19] for plant models.

A test feature model has been proposed, but it has to be specified which is the data needed by the test architecture generator, as well as the optimal way of tracing components of the SUT with test cases. In addition, different modifications in the source code of the FeatureIDE tool might be needed to adapt the plug-in to our needs.

VI. Conclusions and Future Work

This paper presented an automatically configurable test architecture that includes different sources to efficiently and systematically test variability-intensive CPSs. The test architecture is able of self-adapting to product configurations and automatically tests and obtains verdicts that represent results of the executed test cases.

We have focused our approach for the validation of variability-intensive or highly configurable CPSs. As the SUT must deal with variability, the test architecture also has to handle variability. The first step taken has been the detection of the different variation points that the components of the proposed test architecture must deal with. Systems with many variants must use a tool to manage the variability. We have proposed to use Feature Models to manage variability as well as for the traceability among components of the SUT and test cases.

A process with seven steps that enables testing variability-intensive CPSs in a systematic manner has been proposed, starting from variability management with feature modelling and ending with an upload of test results in the test historic database.

This paper has focused on the test architecture and its configurability. Although the test architecture plays an important role when testing highly configurable CPSs, it is also important to study the selection of the appropriate test cases. After developing the framework, it is expected to validate the whole process with different case-studies, desirably from the automotive domain. Additionally, we would like to compare and combine our variability modelling approach with other potential approaches, such as Delta Simulink [24] in test architectures.

ACKNOWLEDGEMENTS

This work has been developed by the embedded systems group supported by the Department of Education, Universities and Research of the Basque Government.

REFERENCES

[1] T. Yue, S. Ali, and K. Nie, "Towards a search-based interactive configuration of cyber physical system product lines," in ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems, Poster, 2013, pp. 71–75.

[2] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," Proceedings of the IEEE (special issue on CPS), vol. 100, no. 1, January 2011, pp. 13 – 28.

[3] J. V. Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in Proceedings of the Working IEEE/IFIP Conference on Software Architecture, ser. WICSA '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 45–54.

[4] S. Thiel and A. Hein, "Systematic integration of variability into product line architecture design," in SPLC, 2002, pp. 130–153.

[5] M. Blackburn, R. Busser, and A. Nauman, "Why model-based test automation is different and what you should know to get started," in In International Conference on Practical Software Quality and Testing, 2004, pp. 87–90.

[6] H. Shokry and M. Hinchey, "Model-based verification of embedded software," Computer, vol. 42, no. 4, 2009, pp. 53 – 59.

[7] T. Berger et al., "A survey of variability modeling in industrial practice," in Variability Modelling of Software-intensive Systems (VaMoS), 2013, pp. 7:1–7:8.

[8] J. Weiland and P. Manhart, "A classification of modeling variability in simulink," in Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, ser. VaMoS '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:8.

[9] G. Sagardui, L. Etxeberria, and J. A. Agirre, "Variability management in testing architectures for embedded control systems," in VALID 2012 - 4th International Conference on Advances in System Testing and Validation Lifecycle, Lisbon, Portugal, 2012, pp. 73 – 78.

[10] Y. Nishi, "Viewpoint-based test architecture design," in 2012 IEEE Sixth International Conference on Software Security and Reliability Companion, Gaithersburg, MD, USA, 2012, pp. 194 – 197.

[11] J. Zander-Nowicka, "Model-based testing of real-time embedded systems in the automotive domain," Ph.D. dissertation, Technical University Berlin, 2008.

[12] A. Arrieta, I. Agirre, and A. Alberdi, "Testing architecture with variability management in embedded distributed systems," in IV Jornadas de Computación Empotrada, ser. JCE 2013, 2013, pp. 12–19.

[13] B. Magro, J. Garbajosa, and J. Perez, "A software product line definition for validation environments," in 12th International Software Product Line Conference (SPLC), Piscataway, NJ, USA, 2008, pp. 45 – 54.

[14] S. Oster, I. Zorcic, F. Markert, and M. Lochau, "Moso-polite - tool support for pairwise and model-based software product line testing," in VaMoS, 2011, pp. 79–82.

[15] C. H. P. Kim et al.,"Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 257–267.

[16] S. Wang, S. Ali, and A. Gotlieb, "Minimizing test suites in software product lines using weight-based genetic algorithms," in Proceedings of the 2013 Genetic and Evolutionary Computation Conference, Amsterdam, Netherlands, 2013, pp. 1493 – 1500.

[17] S. Weißleder and H. Lackner, "Top-down and bottom-up approach for model-based testing of product lines," in MBT, 2013, pp. 82–94.

[18] M. Z. Iqbal, A. Arcuri, and L. Briand, "Empirical investigation of search algorithms for environment model-based testing of real-time embedded software," in Proceedings of the 2012 International Symposium on Software Testing and Analysis, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 199–209.

[19] A. Arrieta, G. Sagardui, and L. Etxeberria, "Towards the automatic generation and management of plant models for the validation of highly configurable cyber-physical systems," in Proceedings of 2014 IEEE 19th Conference on Emerging Technologies & Factory Automation, ser. ETFA 2014, no. (To be published), 2014.

[20] D. Benavides, S. Segura, and A. Ruiz-Corts, "Automated analysis of feature models 20 years later: A literature review," Information Systems, vol. 35, no. 6, 2010, pp. 615 – 636.

[21] C. Dziobek, J. Loew, W. Przystas, and J. Weiland, "Functional variants handling in simulink models," Mathworks, Tech. Rep., 2008.

[22] Pure-Systems. pure::variants. http://www.pure-systems.com. http://www.pure-systems.com. [retrieved: July, 2014]

[23] G. Botterweck, A. Polzer, and S. Kowalewski, "Variability and evolution in model-based engineering of embedded systems," in Tagungsband - Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI, MBEES 2010, Munich, Germany, 2010, pp. 87 – 96.

[24] A. Haber et al., "First-class variability modeling in matlab/simulink," in ACM International Conference Proceeding Series, 2013, pp. 4:1–4:8.

[25] A. Arrieta, G. Sagardui, and L. Etxeberria, "A comparative on variability modelling and management approach in simulink for embedded systems," in V Jornadas de Computación Empotrada, ser. JCE 2014, no. (To be published), 2014.

[26] T. Thuem et al., "Featureide: An extensible framework for feature-oriented software development," Science of Computer Programming, vol. 79, 2014, pp. 70 – 85.

IBM SAN Distance Matrix Project

Trace coverage and modeling across IBM test labs world-wide

Yoram Adler
IBM Research
Haifa, Israel
adler@il.ibm.com

Tara Astigarraga
IBM Corporate Head Quarters
Rochester, United States
asti@us.ibm.com

Sheri Jackson
IBM Systems and Technology Group
Tucson, United States
sheribj@us.ibm.com

Jose Roberto Mosqueda Mejia
IBM Software Group
Guadalajara, Mexico
mosqueda@mx1.ibm.com

Orna Raz
IBM Research
Haifa, Israel
ornar@il.ibm.com

***Abstract*—Storage Area Networks (SAN) solutions are highly complex, often with enterprise class quality requirements. To perform end-to-end customer-like SAN testing, multiple complex interoperability test labs are necessary. One key factor in field quality is test coverage; in distributed test environments this requires a centralized view and coverage model across the different areas of test. We define centralized coverage models and apply our novel trace coverage technology to automatically populate these models. Early results indicate that we are able to create a centralized view of SAN coverage across the multitude of IBM test labs world-wide. Moreover, we are able to compare test lab coverage models with customer environments. Based on these views and comparisons, we expect to obtain an increased coverage, resulting in increased discovery rate of high-impact defects.**

Keywords-Software Test; Software Engineering; SAN Test; System Test; Distance Matrix; Trace Coverage Models; SAN Hardware Test Coverage

I. Introduction and Motivation

IBM is a global technology and innovation company with more than 400,000 employees serving clients in 170 countries [1]. The IBM test structure consists of thousands of test engineers world-wide. In addition to function test teams for product streams, there is also an entire world-wide organization of many hundreds of people dedicated to systems and solution test. IBM has interop and complex test labs world-wide [2]. Systems test strategies focus on customer-like, end-to-end solution integration testing designed to cover the architectural design points of a broad range of customer environments and operations with the end goal of increased early discovery of high-impact defects, resulting in increased quality solutions. One key area of systems and solution test is innovation. As configurations supported continue to climb, with over 180 million configurations supported on the System Storage Interoperation Center (SSIC) site, test engineers are continually challenged to find ways to test smarter [3].

One IBM test transformational project we have been working on is the storage area network (SAN) distance matrix project. This project arose from the IBM Test and Research divisions as a joint-project aimed at better quantifying and understanding the systems test SAN coverage across IBM test groups world-wide.

At the start of this project we had lots of questions related to world-wide hardware and SAN coverage, but we did not have a centralized view of the test labs across IBM. Test labs were built, monitored and architected on an individual basis without the ability to easily extract coverage models across the test locations and understand on a global scale the total IBM coverage model. Another piece missing was the ability to do broad coverage reviews looking at IBM test labs in comparison to its clients. We have always worked hard to build our test environments to include key characteristics from a diverse range of IBM clients, however we did not have a data environment modeling tool to take customer environment variables and map them against our test environments. The IBM Distance Matrix project was designed to address these concerns and help to centralize visibility and configuration details about the systems and solution SAN test labs across IBM and its clients.

The SAN distance matrix project has the abilities to look at key architectural design points across the SAN environments and extract coverage summaries for deep-dive reviews, comparisons and ultimately architecture changes to continually improve our solution test coverage, scalability and customer focus. In this paper, we will further describe the SAN distance matrix project and the early results we have achieved.

II. Related work

There are existing tools, including Cisco Data Center Network Manager [4] and Brocade Network Advisor [5] that provide in-depth and detailed modeling capabilities for single environments or environments managed by a single entity; however there is a gap in the ability to easily look across a heterogeneous group of environments controlled by different companies, divisions or organizations.

In functional modeling and one of its optimization techniques Combinatorial Test Design (CTD), the system under test is modeled as a set of parameters, respective values, and restrictions on value combinations that may not appear together in a test. A test in this setting is a tuple in which every parameter gets a single value. A combinatorial algorithm is applied in order to come up with a test plan (a set of tests) that covers all required interactions between parameters. Kuhn, Wallace and Gallo [6] conducted an empirical study on the interactions that cause faults in software that is the basis for the rationale behind CTD. Nie and Leung [7] provide a recent survey on CTD. The SAN distance matrix that we create can be viewed as a functional model. In our case, we automatically extract the model from switch dumps. We term the creation of coverage models from existing traces 'trace coverage'.

III. Project Strategy

The project strategy is composed of 2 main phases as shown on Fig.1. The goal of the SAN distance matrix project is to extract data quarterly across team world-wide. By identifying key switch data, the script we execute has little impact to the regular activity of the switches. In the following sections, we describe each phase and activities in detail.

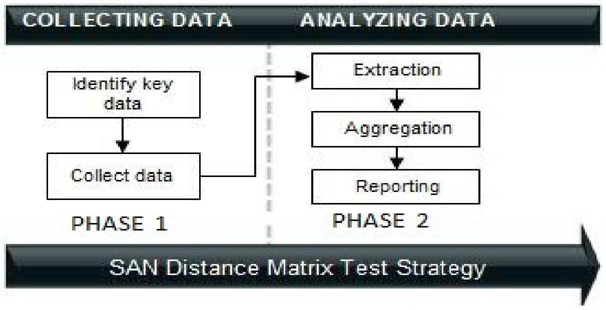


Figure 1. SAN Distance Matrix Project Strategy

IV. Collecting Data

The data collection phase is composed of 2 main activities:

A. *Identify Key Data*

Using switch dump data, we selected specific switch query commands which are used to systematically extract the key data for usage and coverage statistics across different IBM test teams and select customers. The switch query commands allow us to extract dump data focused on topologies, coverage points, utilization and other environmental aspects in our SANs. Topology data points include port speeds, port counts and port types. Environmental data points include the switch hardware platforms, protocols used (ex: Fibre Channel, Fibre Channel over Ethernet, Fibre Channel over IP), code levels, switch up-time and switch special functions/features enabled.

Architectural design points include port-channel/trunk usage, vsan/vlan coverage, virtualization data and initiator/target to inter-switch link ratios. Using this dump data and subsequent processing logic, we were able to create a summary of all the different port speeds being tested, switch utilization rates, general architecture modeling and software and hardware versions being covered across the initial scope of IBM systems test and customer environments.

This approach helped us easily gather promising data, avoid limitations of manual investigation and create a model that is scalable and easy to use for ongoing analysis and trending.

B. *Collect Data*

For data collection, we designed automated scripts to collect the dump data from IBM test labs; the scripts use a source csv file which contains the list of switches, switch types, IPs and credentials. It uses a telnet connection to login the different switches, then executes the appropriate switch query commands and generates a log containing the switch dump data for each switch. For the initial scope of this project a subset of IBM test labs was chosen, that subset group included fourteen IBM test labs which contained a total of four hundred and eighty five SAN top of rack, edge and core switches.

V. Analyzing the Data

The problem: SAN switch dump data is heterogeneous based on switch vendor, platform and code levels. Further, the data is collected from various sources and unique collection methods across IBM test laboratories and customer locations.

The switch dump data is a text file created for each switch. It contains output from multiple switch queries/commands that are executed against the switch. Each switch type has its own set of commands and a unique output format.

The goal: Parse the various switch dump semi structured data and transfer it to structured format.

The solution: the solution relies on the novel notion of trace coverage and the IBM EASER [8] easy log search tool.

Trace coverage extracts report data from traces that already exist in a system or are easy to create according to a defined coverage model. The coverage model can be code coverage – automatically created from the code locations that emit trace data, or functional coverage – manually created to define the system configuration or behavior. In SAN coverage, the traces are created by switch dumps, and the coverage model is a functional coverage of the possible SAN environments. A functional coverage model describes the test space in terms of variation points or attributes and their values. For example, attributes may be port types, ports rates, and ports utilization percentages. The IBM EASER tool supports extraction of semi-structured data from traces and its transformation into structured format. It provides both a graphical user interface (GUI) for interactive exploration and a headless mode of operation for automating the extraction and analysis process.

After defining a functional coverage model, the IBM EASER tool is used to extract, aggregate, and compare data:

- Extract functional model values from switch dumps.
- Aggregate the coverage of multiple logs from both customers and IBM test laboratories.
- Compare coverage between a defined set and subsets of labs by generating multiple summary reports.

The SAN Test functional coverage model is extendable; it can be updated to include additional values seen in customer environments. The collected data is aggregated by

IBM test groups and customers and definitions are flexible and can be supplied by the end-user.

The automated functional coverage analysis process includes three phases: Extraction, Aggregation and Reporting.

A. Extraction

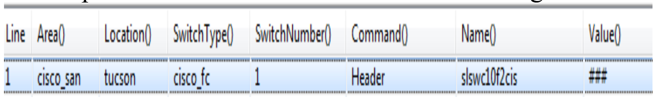
The functional model attributes' values are extracted from each switch dump file. By using EASER, the log is divided into entries and then the relevant data is extracted, computed and inserted into the relevant model attributes' values. One file with attributes and values is created for each switch log file.

Fig. 2 shows a sample of a single cisco_fc switch log file, which is created using the automated scripts. In addition to the switch summary, the log file includes the switch query commands and corresponding switch data output.

```
2014-03-24 14:24:55 INFO Switch Summary
    Name:    slswc10f2cis
    IPAddr:   9.11.195.75
    Brand:   cisco
    Type:     fc
    Area:     cisco san
    Location: tucson
2014-03-24 14:24:55 INFO Log in to  device slswc10f2cis.tuc.stglabs.ibm.com
2014-03-24 14:25:00 INFO Log in to slswc10f2cis.tuc.stglabs.ibm.com successful
2014-03-24 14:25:00 INFO --------------------------------------------------
```

Figure 2. Switch Log File Sample

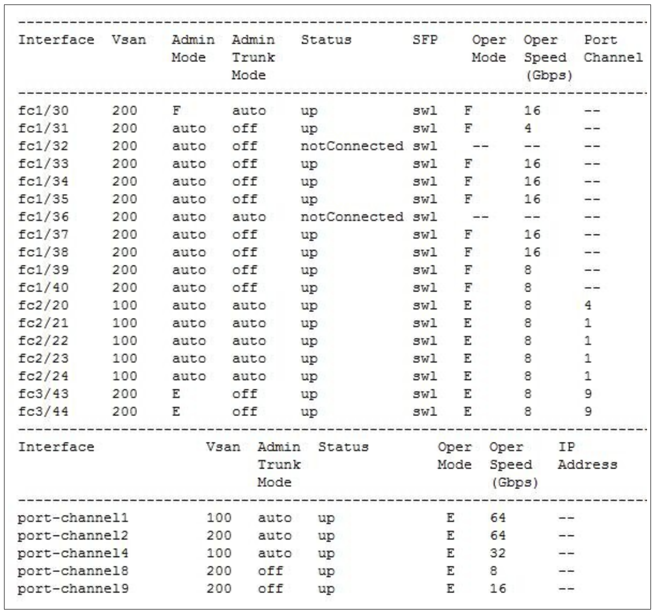
The EASER parser extracts values from the entry in Fig. 2 and updates them into the attributes shown in Fig 3.



Line	Area()	Location()	SwitchType()	SwitchNumber()	Command()	Name()	Value()
1	cisco_san	tucson	cisco_fc	1	Header	slswc10f2cis	###

Figure 3. Parser extracted data Sample

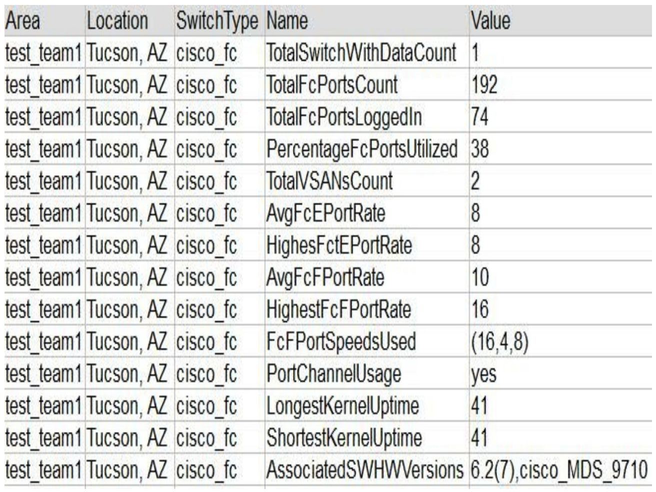
Fig. 4, shows a sample of a Cisco switch dump data extract, which the parser will use to compute values, then inserts the combined values into a model.



```
--------------------------------------------------------------------------------
Interface  Vsan   Admin  Admin   Status        SFP    Oper  Oper   Port
                  Mode   Trunk                        Mode  Speed  Channel
                         Mode                               (Gbps)
--------------------------------------------------------------------------------
fc1/30     200    F      auto    up            swl    F     16     --
fc1/31     200    auto   off     up            swl    F     4      --
fc1/32     200    auto   off     notConnected  swl     --   --     --
fc1/33     200    auto   off     up            swl    F     16     --
fc1/34     200    auto   off     up            swl    F     16     --
fc1/35     200    auto   off     up            swl    F     16     --
fc1/36     200    auto   auto    notConnected  swl     --   --     --
fc1/37     200    auto   off     up            swl    F     16     --
fc1/38     200    auto   off     up            swl    F     16     --
fc1/39     200    auto   off     up            swl    F     8      --
fc1/40     200    auto   off     up            swl    F     8      --
fc2/20     100    auto   auto    up            swl    E     8      4
fc2/21     100    auto   auto    up            swl    E     8      1
fc2/22     100    auto   auto    up            swl    E     8      1
fc2/23     100    auto   auto    up            swl    E     8      1
fc2/24     100    auto   auto    up            swl    E     8      1
fc3/43     200    E      off     up            swl    E     8      9
fc3/44     200    E      off     up            swl    E     8      9
--------------------------------------------------------------------------------
Interface                Vsan  Admin  Status          Oper  Oper    IP
                               Trunk                  Mode  Speed   Address
                               Mode                         (Gbps)
--------------------------------------------------------------------------------
port-channel1            100   auto   up              E     64      --
port-channel2            200   auto   up              E     64      --
port-channel4            100   auto   up              E     32      --
port-channel8            200   off    up              E     8       --
port-channel9            200   off    up              E     16      --
```

Figure 4. Cisco MDS extract data snippet

Fig. 5 shows an example of an abbreviated model. For the sake of brevity, only a small portion of the parser extract and model data are shown in these figures.



Area	Location	SwitchType	Name	Value
test_team1	Tucson, AZ	cisco_fc	TotalSwitchWithDataCount	1
test_team1	Tucson, AZ	cisco_fc	TotalFcPortsCount	192
test_team1	Tucson, AZ	cisco_fc	TotalFcPortsLoggedIn	74
test_team1	Tucson, AZ	cisco_fc	PercentageFcPortsUtilized	38
test_team1	Tucson, AZ	cisco_fc	TotalVSANsCount	2
test_team1	Tucson, AZ	cisco_fc	AvgFcEPortRate	8
test_team1	Tucson, AZ	cisco_fc	HighesFctEPortRate	8
test_team1	Tucson, AZ	cisco_fc	AvgFcFPortRate	10
test_team1	Tucson, AZ	cisco_fc	HighestFcFPortRate	16
test_team1	Tucson, AZ	cisco_fc	FcFPortSpeedsUsed	(16,4,8)
test_team1	Tucson, AZ	cisco_fc	PortChannelUsage	yes
test_team1	Tucson, AZ	cisco_fc	LongestKernelUptime	41
test_team1	Tucson, AZ	cisco_fc	ShortestKernelUptime	41
test_team1	Tucson, AZ	cisco_fc	AssociatedSWHWVersions	6.2(7),cisco_MDS_9710

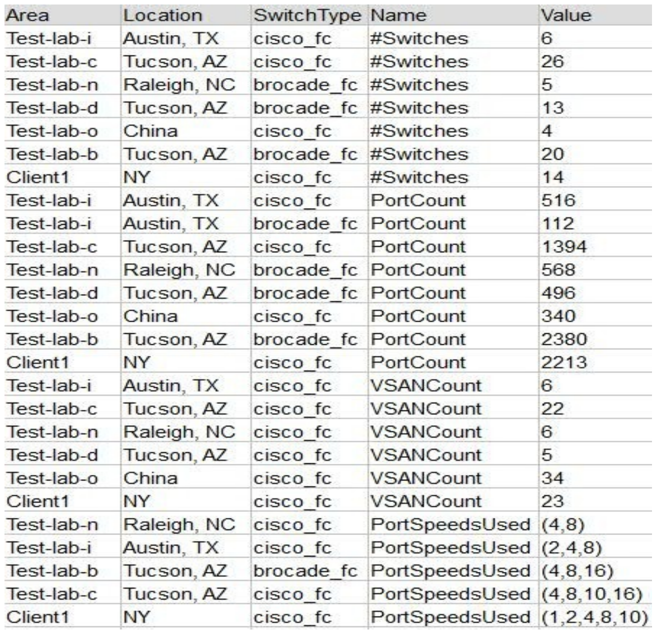
Figure 5. Cisco MDS single switch abbreviated base model.

B. Aggregation

All data from **Extraction** output files is grouped by switch type and switch locations into three files:

1. Summary of all entries,
2. Summary of all samples that contains "full data"
3. Summary of files with "no" or "partial" data.

The contents of the first two files reflect the model: Attributes and their aggregated values from the extraction phase output files. The third file contains an 'illegal' list that should be reviewed by IBM experts for the cause of the failure during collection. Fig. 6 contains a subset example.



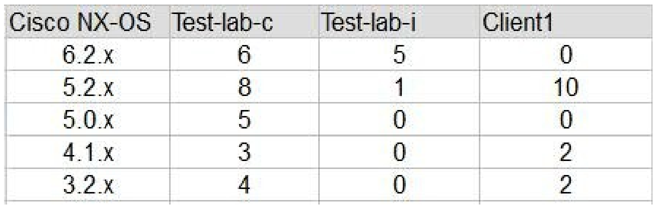
Area	Location	SwitchType	Name	Value
Test-lab-i	Austin, TX	cisco_fc	#Switches	6
Test-lab-c	Tucson, AZ	cisco_fc	#Switches	26
Test-lab-n	Raleigh, NC	brocade_fc	#Switches	5
Test-lab-d	Tucson, AZ	brocade_fc	#Switches	13
Test-lab-o	China	cisco_fc	#Switches	4
Test-lab-b	Tucson, AZ	brocade_fc	#Switches	20
Client1	NY	cisco_fc	#Switches	14
Test-lab-i	Austin, TX	cisco_fc	PortCount	516
Test-lab-i	Austin, TX	brocade_fc	PortCount	112
Test-lab-c	Tucson, AZ	cisco_fc	PortCount	1394
Test-lab-n	Raleigh, NC	brocade_fc	PortCount	568
Test-lab-d	Tucson, AZ	brocade_fc	PortCount	496
Test-lab-o	China	cisco_fc	PortCount	340
Test-lab-b	Tucson, AZ	brocade_fc	PortCount	2380
Client1	NY	cisco_fc	PortCount	2213
Test-lab-i	Austin, TX	cisco_fc	VSANCount	6
Test-lab-c	Tucson, AZ	cisco_fc	VSANCount	22
Test-lab-n	Raleigh, NC	cisco_fc	VSANCount	6
Test-lab-d	Tucson, AZ	cisco_fc	VSANCount	5
Test-lab-o	China	cisco_fc	VSANCount	34
Client1	NY	cisco_fc	VSANCount	23
Test-lab-n	Raleigh, NC	cisco_fc	PortSpeedsUsed	(4,8)
Test-lab-i	Austin, TX	cisco_fc	PortSpeedsUsed	(2,4,8)
Test-lab-b	Tucson, AZ	brocade_fc	PortSpeedsUsed	(4,8,16)
Test-lab-c	Tucson, AZ	cisco_fc	PortSpeedsUsed	(4,8,10,16)
Client1	NY	cisco_fc	PortSpeedsUsed	(1,2,4,8,10)

Figure 6. Summary of select full data samples

C. Reporting

Data from the **Aggregation** phase is broken into several reports. There are two summary reports types: code levels and machine types which are based on aggregation summary of all entry files and results report which contains data including: switch functions, SAN design principles, switch utilization, port speeds, errors, peak traffic rates and average traffic rates.

Fig 7. contains an example of number of switches running select Cisco NX-OS code levels from two IBM test labs and one client location.



Cisco NX-OS	Test-lab-c	Test-lab-i	Client1
6.2.x	6	5	0
5.2.x	8	1	10
5.0.x	5	0	0
4.1.x	3	0	2
3.2.x	4	0	2

Figure 7. Code level sample report

VI. Early Results

We have created a functional model that has enabled us to:

- Identify key SAN coverage and test variants
- Better understand our global SAN test environments using trace coverage analysis to aid in gap identification and improve our system test coverage strategies.
- Ensure test groups remain on IBM supported Cisco and Brocade switch code levels
- Create central list of SAN switch hardware across IBM test, allowing us to identify groups utilizing dated switch hardware and place them into a hardware refresh pool.
- Look at switch utilization and stress rates to ensure we are accurately stressing our equipment and if not put plans in place to help increase load coverage.
- Review environment architecture designs and recommend changes or complexity additions where appropriate.
- Foster technical interaction and deep dive environment cross-test-cell reviews with test technical leads from IBM test labs world-wide.

Overall, we were able to systematically collect data from global IBM System test labs and create a centralized view of SAN switch equipment and coverage across IBM systems test. We were also able to gather dump data from select customers and compare our test lab coverage models with customer environments.

VII. Conclusion and Further Development

As solution complexity and the number of supported configurations increase in the IT industry, we must continue to re-invent the ways we do solution testing. In our global test environment, the need to have procedures in place to extract data and create advanced comparison and coverage models is essential. This project, although in its early deployment stages, has already shown tremendous promise for being able to systematically extract and model coverage across a large number of test and client SAN environments. One of the key factors of this models continuing success is its scalability.

In the future, we plan to extend the distance function beyond reducing the data to a single dimension. For example, today one distance function is the difference in the average rates among different groups. We could instead compute a distance metric over the rate vectors.

As we continue to implement the distance matrix project across test labs within IBM we are gathering key data and making methodical changes is SAN test architecture to provide better test coverage points for IBM products and solutions.

References

[1] “IBM Basics,” ibm.com [Online]. Available from: http://www.ibm.com/ibm/responsibility/basics.shtml. [Accessed: May 15, 2014].

[2] T. Astigarraga, “IBM Test Overview and Best Practices” SoftNet 2012, Available from: http://www.iaria.org/conferences2012/filesVALID12/IBM_Test_Tutorial_VALID2012.pdf [Accessed: May 15, 2014].

[3] “IBM System Storage Interoperation Center (SSIC),” ibm.com [Online]. Available from: http://www-03.ibm.com/systems/support/storage/ssic/interoperability.wss [Accessed: May 15, 2014].

[4] “Cisco DCNM Overview,” cisco.com [Online]. Available from: http://www.cisco.com/c/en/us/td/docs/switches/datacenter/mds9000/sw/5_2/configuration/guides/fund/DCNM-SAN-LAN_5_2/DCNM_Fundamentals/fmfundov.html [Accessed: May 28, 2014].

[5] “Brocade Network Advisor,” brocade.com [Online]. Available from: http://www.brocade.com/products/all/management-software/product-details/network-advisor/index.page [Accessed: May 28, 2014]

[6] Kuhn, D. Richard, Dolores R. Wallace, and Jr AM Gallo. "Software fault interactions and implications for software testing." Software Engineering, IEEE Transactions on 30.6 (2004): pp. 418-421

[7] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. ACM Comput. Surv. 43, 2, Article 11 (February 2011)

[8] Y. Adler, A. Aradi, Y. Magid, O. Raz, “IBM Log Analysis Tool (EASER),” unpublished.