VALID 2015

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

ISBN: 978-1-61208-441-1

November 15 - 20, 2015

Barcelona, Spain

VALID 2015 Editors

Teemu Kanstren, VTT, Finland

Birgit Gersbeck-Schierholz, Leibniz Universität Hannover, Germany

VALID 2015

Forward

The Seventh International Conference on Advances in System Testing and Validation Lifecycle (VALID 2013), held on November 15 - 20, 2015 in Barcelona, Spain, continued a series of events focusing on designing robust components and systems with testability for various features of behavior and interconnection.

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

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

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

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

We take here the opportunity to warmly thank all the members of the VALID 2015 technical program committee as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and efforts to contribute to VALID 2015. We truly believe that thanks to all these efforts, the final conference program consists of top quality contributions.

This event could also not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the VALID 2015 organizing committee for their help in handling the logistics and for their work that is making this professional meeting a success. We gratefully appreciate to the technical program committee co-chairs that contributed to identify the appropriate groups to submit contributions.

We hope the VALID 2015 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in system testing and validation. We also hope Barcelona provided a pleasant environment during the conference and everyone saved some time for exploring this beautiful city.

VALID 2015 Advisory Chairs

Andrea Baruzzo, Università degli Studi di Udine, Italy
Cristina Seceleanu, Mälardalen University, Sweden

Mehdi Tahoori, Karlsruhe Institute of Technology (KIT), Germany
Mehmet Aksit, University of Twente - Enschede, The Netherlands
Amir Alimohammad, San Diego State University, USA

VALID 2015 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, Gunma University, 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 2015 Industry Chairs

Abel Marrero, Bombardier Transportation Germany GmbH - Mannheim, Germany

VALID 2015

Committee

VALID Advisory Chairs

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

VALID 2015 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, Gunma University, 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 2015 Industry Chairs

Abel Marrero, Bombardier Transportation Germany GmbH - Mannheim, Germany

VALID 2015 Technical Progam Committee

Fredrik Abbors, Åbo Akademi University, Finland
Jaume Abella, Barcelona Supercomputing Center (BSC-CNS), Spain
Mehmet Aksit, University of Twente - Enschede, The Netherlands
Amir Alimohammad, San Diego State University, USA
Giner Alor Hernandez, Instituto Tecnologico de Orizaba - Veracruz, México
César Andrés Sanchez, Universidad Complutense de Madrid, Spain
Aitor Arrieta, Mondragon Unibertsitatea, Spain
Selma Azaiz, CEA List Institute - Gif-Sur-Yvette, France
Cesare Bartolini, ISTI - CNR, Pisa, Italy
Andrea Baruzzo, Università degli Studi di Udine, Italy
Serge Bernard, LIRMM, Franmce
Paolo Bernardi, Politecnico di Torino, Italy
Ateet Bhalla, Independent Consultant, India
Mauro Birattari, Université Libre de Bruxelles, Belgium
Bruno Blaškovic, Faculty of Electrical Engineering and Computing ZOEEM - CRS lab, Croatia
Mark Burgin, University of California Los Angeles (UCLA), USA
Isabel Cafezeiro, Instituto de Computação - Universidade Federal Fluminense, Brazil
Luca Cassano, University of Pisa, Italy

Henrique Rebêlo, Federal University of Pernambuco, Brazil
Eike Reetz, University of Applied Sciences Osnabrück, Germany
Oliviero Riganelli, University of Lugano, Switzerland
Goiuria Sagardui Mendieta, Mondragon University, Spain
Christian Schanes, Vienna University of Technology, Austria
Cristina Seceleanu, Mälardalen University, Sweden
Nassim Seghir, University of Oxford, UK
Sergio Segura, University of Seville, Spain
Vipul Shah, Tata Consultancy Services, India
Alin Stefanescu, University of Bucharest, Romania
Mehdi B. Tahoori, Karlsruhe Institute of Technology (KIT), Germany
Nur A. Touba, University of Texas - Austin, USA
Spyros Tragoudas, Southern Illinois University Carbondale, USA
Jos van Rooyen, Bartosz ICT, Netherlands
Miroslav N. Velev, Aries Design Automation, USA
R. Venkatesh, Tata Consultancy Services, India
Bart Vermeulen, NXP Semiconductors - Eindhoven, The Netherlands
Arnaud Virazel, Université Montpellier 2 / LIRMM, France
Tanja E. J. Vos, Universidad Politécnica de Valencia, Spain
Stefan Wagner, University of Stuttgart, Germany
Hironori Washizaki, Waseda University, Japan
Kristian Wiklund, Ericsson AB / Mälardalen University, Sweden
Lina Ye, CentraleSupélec, Gif sur Yvette, France / LRI, Univ. Paris-Sud 11, France
Cemal Yilmaz, Sabanci University - Istanbul, Turkey
Zeljko Zilic, McGill University, Canada

Table of Contents

An Experimental Comparative Study of Fault-Tolerant Architectures

Imran Wali, Arnaud Virazel, Alberto Bosio, Patrick Girard
LIRMM – University of Montpellier / CNRS
Montpellier, France
e-mail: {wali, virazel, bosio, girard}@lirmm.fr

Abstract—**This paper provides a comparative study based on experiments performed on four similar fault-tolerant architectures intended to reduce errors caused due to faults in combinational logic parts of microelectronic circuits and systems. The compared merits include area, power, performance and fault tolerance capability. The experimental results show that the improved Hybrid Fault-Tolerant Architecture can handle transient faults as effectively as Partial-TMR and exhibits permanent fault tolerance capability similar to that of Full-TMR. It offers 11.8% and 20.5% power saving compared to Partial and Full-TMR respectively. Furthermore, it can handle the fault accumulation effect better than TMR, hence an ideal candidate for low-power long duration mission-critical applications.**

Keywords-fault tolerant architecture; fault tolerance capability assessment.

I. INTRODUCTION

Complementary metal-oxide semiconductor (CMOS) device scaling is posing reliability challenges to future microelectronic circuits and systems [1]. Other alternative and evolutionary technologies are also facing reliability issues in their early development life cycles. Design architects must address the concern of preventing reliability from becoming a bottleneck for the development of high-performance, low-power systems, through the use of fault-tolerant techniques.

These techniques are commonly used to tolerate on-line faults, i.e., faults that appear during the normal functioning of the system, irrespective of their transient or permanent nature [2]. They use redundancy, i.e., the property of having spare resources that perform a given function and tolerate faults in the combinational [3]-[5] and/or sequential [6]-[9] part of the circuit. These techniques are generally classified by the type of redundancy used. Basically, three types of redundancy are considered: information, temporal and hardware [2].

Many studies in literature like [10]-[12] provide evaluation results within the scope of the architecture proposed therein. However, it is essential that these similar schemes be comprehensively compared using identical set of experiments and conditions in order to have a meaningful contrast. For any fault-tolerant architecture, the four merits that are essential to be analyzed are its area, power and performance overheads and most importantly its fault tolerance capability. Among these four merits area, power and performance can be evaluated using conventional circuit analysis tools. Unlike these attributes of a fault-tolerant architecture, fault tolerance capability cannot be evaluated using standard circuit analysis methodologies, but only by observing system behavior in the presence of faults [13].

In this paper, we present a comprehensive experimental comparative study of four fault-tolerant architectures with similar fault-tolerance capability in the context of spatial and temporal characteristics of faults and the architectural cost merits, which include area and power consumption. These architectures include Partial Triple Modular Redundancy (Partial-TMR) and Full Triple Modular Redundancy (Full-TMR) [2], Hybrid Fault-Tolerant (HyFT) [14][15] and improved Hybrid Fault-Tolerant (*i*HyFT) [16] architectures. For assessing the merits of these fault-tolerant architectures, we implement them on some ITC'99 benchmarks and use a Gate-level simulation based fault-injection framework to quantitatively assess and compare the fault tolerance capability of these schemes.

The remaining parts of this paper are organized as follows. Section 2 highlights the problematic of error occurrences in combinational logics and storage elements. Section 3 presents the fault-tolerant architectures under comparison. Section 4 details the experimental methodology while Section 5 gives results in terms of area, power, performance and fault-tolerance capability. Finally, Section 6 concludes the paper and provides some perspectives.

II. PROBLEM STATEMENT

Lidén et al. in 1994 experimentally estimated that only 2% of bit flips in memory elements also known as Single Event Upset (SEU) were caused by particle-induced transients or Single Event Transients (SET) generated in and propagated through Combinational Logic (CL). The rest were due to direct particle strike in latches. Their experiments involved using a 1μm CMOS process at 5MHz [17]. Since then physical gate-length has downscaled up to 50 times, supply voltages have dropped to 0.9 V and operating frequency has shown a thousand fold increase [1]. This massive change in technology has resulted in greater sensitivity of memory elements to high-energy particle, but the effects are more pronounced on CL networks [18]. A more recent work uses a probability model to estimate that the susceptibility to CL circuits to SET nearly doubles as the technology scales from 45 nm to 16 nm [19]. As a result research attention drawn towards developing techniques to limit Soft Error Rate (SER) in CL is becoming comparable to effort made in protecting state elements. Figure 1 symbolically illustrates the share and types of problems arising from sequential logic and combinational logic parts of digital circuit.

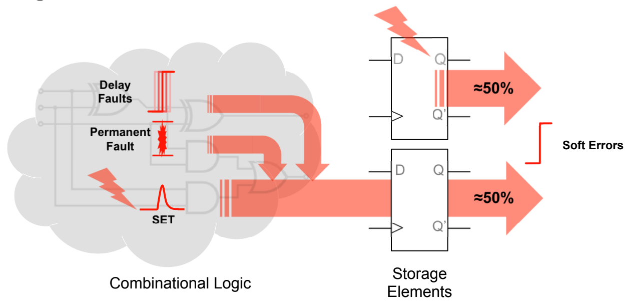
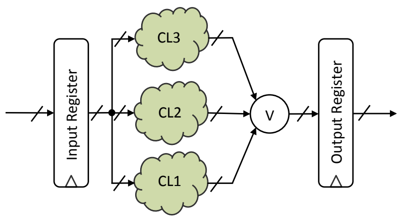


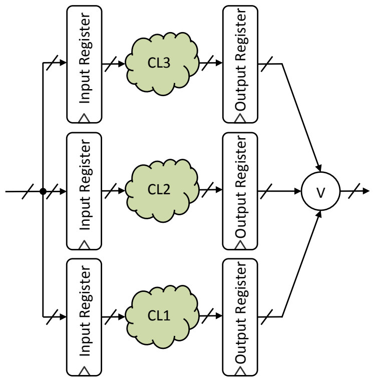
Figure 1. Error occurrences in combinational logics and storage elements

III. Fault-Tolerant Architectures

Several hardware fault-tolerant architectures have been proposed in the literature [20]. The classical hardware redundancy architecture is the N Modular Redundancy (NMR). A NMR structure is a fault-tolerant architecture based on *N* modules performing the same function. The outputs of these modules are compared by using a majority voter. The case of $N = 3$ is called TMR and has been widely studied and used in practical system applications [2][3].



(a)



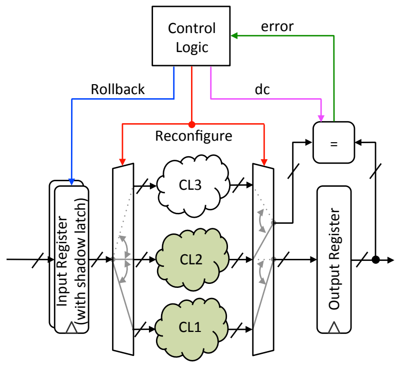
(b)

Figure 2. TMR Architectures (a) Partial-TMR and (b) Full-TMR

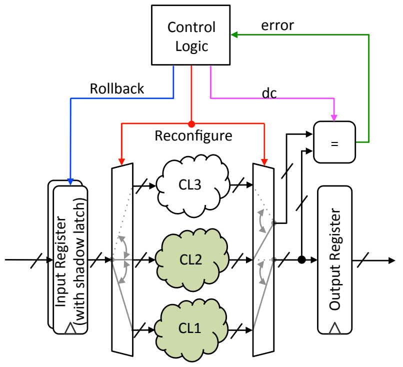
There are different methods to implement TMR architecture for logic circuits, depending on which part of this circuit is triplicated. In Figures 2.a and 2.b, we present two TMR structures that will be compared with the hybrid fault-tolerant architecture. The first implementation (Partial-TMR, Figure 2.a) consists of triplicating only Combinational Logic (CL) part of the logic circuit while the second one (Full-TMR, Figure 2.b) requires triplications of both combinational and sequential parts.

While having smaller area overhead, the partial-TMR solution cannot tolerate SEUs or permanent faults in pipeline registers. This problem can be solved using full-TMR solutions by triplicating the registers. Note that in full-TMR input registers are also triplicated so that errors caused by each register can be tolerated.

The second fault-tolerant architecture under comparison is the HyFT scheme presented in [14, 15]. This architecture employs information redundancy (duplication/comparison) for the error detection, timing redundancy (re-computation) for the transient error correction and hardware redundancy (re-configuration) for the permanent error correction. As presented in Figure 3.a, the hybrid architecture employs three copies of CL (CL1, CL2 and CL3) modules. The input demultiplexer and the output multiplexer are used to select two running CL copies and to put the third CL copy in standby mode.



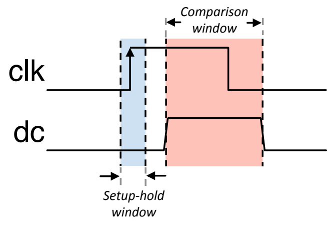
(a)



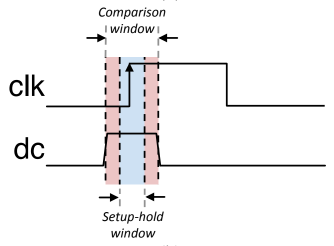
(b)

Figure 3. HyFT Architectures (a) HyFT and (b) *i*HyFT

The HyFT architecture is driven by a control logic module, which is divided in two parts. The first part consists of a state-machine that controls different configurations of the architecture, i.e., it decides which two CL copies to run in parallel. The second part controls the comparator, pipeline register, demultiplexer and multiplexer. For error detection it uses the pseudo-dynamic comparator presented in [21]. It combines a dynamic transition detector and a static comparator in order to detect hard, soft and timing errors during a *comparison-window* as shown in the timing diagram of Figure 4.a. The comparison takes place only during these brief intervals of time represented as red shaded regions with doted outline in Figure 4. The timing of *comparison-window* is defined by the high phase of a delayed clock signal '*dc*'.



(a)



(b)

Figure 4. Comparison-window timing for (a) HyFT and (b) *i*HyFT

Figure 3.b presents the improved version of the HyFT (*i*HyFT) architecture, which resulted as an attempt to improve the error detection capability and to reduce the performance overhead [16]. This improved scheme achieves aforementioned objectives by using a *comparison-window* across the *setup-hold window* as shown in Figure 4.b. With this comparison timing it can intrinsically detect erroneous signal transients that are more likely to be captured in the output register. The *comparison-window* timing was made possible by changing the placement of the comparator such that the comparator compares the output of two running CL copies directly from the multiplexer as shown in Figure 3.b. The ability to act against only the potentially fatal SETs not only reduces the number of fail-silent faults but also improves the performance.

IV. EXPERIMENT METHODOLOGY

Experiments are performed to compare the merits of the four fault-tolerant architectures presented in Section III. Each architecture is applied to a few of the ITC'99 benchmark circuits and are synthesized using NanGate 45nm Open Cell Library [22]. The area figures are obtained from the synthesized designs and power estimates are obtained by taking into account the switching activity generated by back-annotated gate-level simulations. The workload for simulation is a set of patterns optimized for stuck-at fault detection. The reason for using such a workload is to obtain switching activity distributed in all parts of the circuit.

The performance overhead is evaluated in two different aspects. Firstly, in terms of temporal performance degradation, which is basically the additional delay in the data-path due to the fault-tolerant architecture (e.g., voter delay in TMR), and secondly in terms of error recovery penalty under a certain fault rate.

The fault-tolerance capability of the four schemes is estimated by performing fault injection in the combinational logic parts of the circuits, by using a gate-level simulation based fault-injection framework. The framework uses the switching activity file to extract the list of all possible fault-locations. From this list it randomly selects a subset of locations for fault-injection. To each fault location in this subset, it randomly assigns a fault-injection time within the limits of simulation time duration and a SET duration also randomly selected from the range of typically anticipated SET pulses, i.e. from 0.25ns to 1.25ns [23]. Once the fault list is prepared, fault injection campaigns that comprise a number of simulations are run. Either a single SETs or a permanent stuck-at fault is injected per simulation by forcing the signals at the specified location, at the corresponding time indicated by the fault list.

During the fault injection campaign a fault-injection report is generated which contains the cycle-by-cycle outcome of each simulation. At the end of fault-injection campaign the fault-injection report is analyzed to classify the faults according to the fault effects into three categories:

1. **Silent faults**: the faults that have no impact on the workload computation nor are detected by the fault-tolerant architecture.
2. **Corrected faults**: the faults that are detected and corrected by the fault- architecture in place.
3. **Fail-Silent fault**: the faults, which result in a wrong computed result but are not detected by the fault-tolerant architecture.

The ratio of the number of fail-silent faults to the number of total injected faults gives us a figure to compare the fault tolerance capability of the four different schemes.

V. COMPARATIVE ANALYSIS

A. Area and Power Overhead

Table I gives the average area and power for the BaseLine (BL) circuits and the fault-tolerant schemes based on the results of their implementation on six ITC'99 benchmark circuits. It also gives their associated overheads of area and power with the BL circuits as reference.

TABLE I. AREA AND POWER ESTIMATION RESULTS

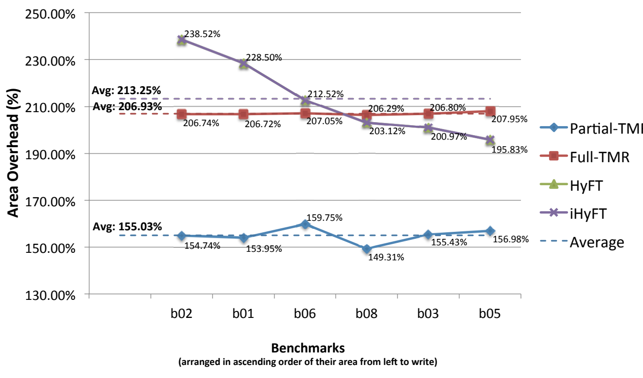
	Avg Area (μm^2)	Avg Area overhead (%)	Avg Power (μW)	Avg Power overhead (%)
BL	1231.00	0	351.50	0
Partial-TMR	3141.59	155.02	971.66	173.32
Full-TMR	3781.32	206.93	1077.74	206.09
HyFT	3739.43	213.24	859.67	157.36
*i*HyFT	3739.43	213.24	856.49	156.83

The most obvious area and power overhead figures are those of Full-TMR. As it is based on triplicating the CL blocks and also the registers, it occupies a little more than three times the area and consumes a few microwatts over BL. This extra area and power is due to the voter in the Full-TMR scheme.

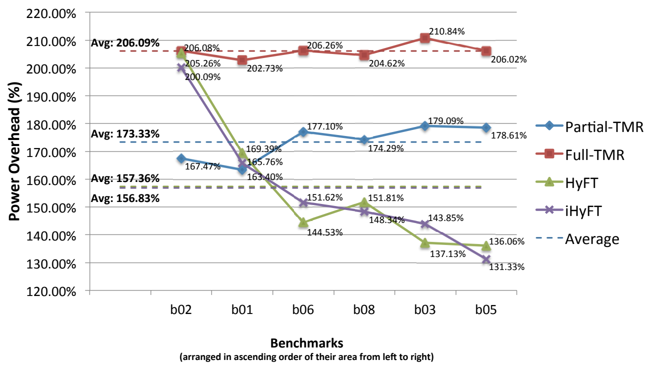
The average percentage of area overhead values in Table I show that the partial-TMR implementation consumes less in terms of area that is about 155%. The two most expensive architectures in terms of area are HyFT and *i*HyFT with an average overhead of around 213% on average for the considered set of benchmark circuits.

As far as the power consumption is concerned HyFT and *i*HyFT architectures are most efficient based on the average power overhead figure of about 157% in Table I. Partial-TMR stands at 173%, making Full-TMR the least power efficient scheme. This high power consumption is accounted to the triplication of sequential elements. On the other hand HyFT and *i*HyFT save power by having one CL copy in stand-by all the time.

The graphs in Figure 5 show the percentage increase in area (Figure 5.a) and power (Figure 5.b) of the BL circuits to implement the four fault-tolerant architectures discussed in Section III. Note that the benchmark circuits are arranged in ascending order of their size from left to right on X-axis to illustrate the impact of the size of CL block on the area and power overheads. The dotted lines in Figure 5 represent the average percentage figures of area and power overheads for the corresponding fault-tolerant architecture implementation.



(a)



(b)

Figure 5. Impact of CL block size on (a) Area and (b) Power Overhead

An important observation that can be made in the graphs of Figure 5 is that, the area and power overheads of both partial and full-TMR are relatively independent of the size of CL block to which they are applied. However, these overheads for HyFT and *i*HyFT change with different sizes of benchmarks such that the area and power overheads of HyFT and *i*HyFT decrease with the larger benchmarks. This observation also gives an idea of the anticipated impact on the area and power overheads for CL blocks larger than the benchmarks considered in this study. Although the average area overhead of HyFT and *i*HyFT is higher than other considered fault-tolerant architectures but with large CL blocks we can expect it to decrease. Where as the power overhead of HyFT and *i*HyFT, which is already the minimum, tends to further reduce with larger CL blocks.

B. *Performance*

The first evaluated measure of performance is the temporal performance degradation. In partial-TMR and full-TMR, it is defined by the delay of voter circuit in the data-path. In case of HyFT and *i*HyFT it is due to the delay of shadow latch multiplexers in input register responsible for rollback and the reconfiguration multiplexer and demultiplexer. The comparator being outside the critical path does not contribute to the temporal performance degradation. Using static timing analysis the temporal performance degradation for partial-TMR and full-TMR was estimated to be 0.73% for a 100MHz operation. The same for HyFT and *i*HyFT was found to be 9.7% without any design optimization.

The figures that can give us a measure of the second considered performance aspect, i.e. the error recovery penalty, can be interpreted from the transient fault injection results presented in Table II. These results are obtained by injecting transient faults at an average rate of 250K faults/second.

TABLE II. TRANSIENT FAULT INJECTION EXPRIMENT RESULTS SUMMARY

	Avg % of Silent faults	*Avg % of Corrected faults*	*Avg % of Fail-silent faults*
BL	92.51%	0.00%	7.49%
Partial-TMR	99.97%	0.00%	0.03%
Full-TMR	100%	0.00%	0.00%
HyFT	92.46%	7.26%	0.28%
*i*HyFT	94.10%	5.86%	0.04%

It can be observed in Table II that for partial-TMR and full-TMR the percentage of corrected faults is zero. This is because; TMR is an error masking technique rather than an error detection and correction one and does not indicate the presence of error. With no provision of identifying the corrected faults, they are kept within the category of silent faults in our analysis. It also indicates that the error recovery penalty for TMR is zero as it corrects errors by masking them instead of undergoing a reconfiguration and re-computation cycle. It can also be seen in Table II that HyFT corrected on average 7.26% of injected faults. For each detected and corrected SET the HyFT undergoes a recovery phase that takes 2 additional cycles [14]. According to these figures, HyFT spends around 14.52% of total computation time on recovering from potentially erroneous states. On the other side, *i*HyFT spends 11.72% of time in recovery phase under the same fault rate.

C. *Fault-tolerance capability*

1) *Quantitative analysis*

To compare the fault-tolerance capability of different architectures we analyze them in terms of the percentage of faults that resulted in a fail-silent outcome among the total number of injected faults. Table III gives the transient fault-injection experiment results.

TABLE III. TRANSIENT FAULT INJECTION RESULTS

	Percentage of fail-silent faults (%)				
	BL	***Partial-TMR***	***Full-TMR***	***HyFT***	***iHyFT***
b01	7.49	0.00	0.00	0.37	0.12
b02	8.11	0.11	0.00	0.28	0.11
b03	8.18	0.03	0.00	0.26	0.03
b05	7.11	0.02	0.00	0.21	0.00
b06	7.07	0.04	0.00	0.33	0.09
b08	7.45	0.05	0.00	0.33	0.08
Average	**7.56%**	**0.03%**	**0.00%**	**0.28%**	**0.04%**

Table III shows that the incorporation of each fault-tolerant architecture into the BL circuit reduces the percentage of fail-silent faults to a different extent. The percentage of fail-silent faults that was originally 7.56% in BL is brought down to 0.03% by partial-TMR. A through analysis of the fault-injection report revealed that these 0.03% faults were among those which were injected at the inputs of CL blocks and effected all the three TMR copies in the same way, thus resulted in a common-mode failure. Full-TMR on the other hand did not encounter this problem because of it construction and turned out to be the most effective by tolerating the effects of all the injected transient faults.

In case of HyFT 0.28% of injected faults escaped detection and effected the results. With further investigation we found out that these fail-silent outcomes were not linked to a specific location as in case of partial-TMR, but escaped detection due to their specific timing characteristics. Static timing analysis showed that these 0.28% fail-silent faults were among those that were injected at a time such that their effects appeared at the inputs of register during the clock *setup-hold window*. Since in HyFT the *comparison-window* does not overlap the *setup-hold window* as discussed in Section III and shown in Figure 4.a, these transient faults managed to affect the data during captured but escaped detection by missing the *comparison-window*. This problem of non-overlapping *setup-hold window* and *comparison-window* was solved by *i*HyFT and therefore significant reduction in the percentage of fail-silent faults is observed in *i*HyFT of about 0.04%.

Similar observations can be made form the permanent fault injection results shown in Table IV. An average 1.36% of faults injected in partial-TMR result in fail-silent outcome, mainly due to the common-mode effect. Full-TMR and *i*HyFT show nearly complete tolerance against permanent faults and in HyFT 0.08% faults escaped detection mainly due to the *setup-hold window* and *comparison-window* separation.

TABLE IV. PERMANENT FAULT INJECTION RESULTS

	Percentage of fail-silent faults (%)				
	BL	***Partial-TMR***	***Full-TMR***	***HyFT***	***iHyFT***
b01	98.37	2.37	0.00	0.15	0.02
b02	96.28	2.03	0.00	0.06	0.00
b03	98.15	1.38	0.00	0.06	0.00
b05	97.84	0.50	0.00	0.08	0.00
b06	97.23	0.66	0.00	0.13	0.00
b08	98.03	2.34	0.00	0.07	0.00
Average	**98.03%**	**1.36%**	**0.00%**	**0.08%**	**0.00%**

2) *Qualitative analysis*

Some aspects of fault-tolerance capability that have an implication on the lifetime reliability of the circuit cannot be inferred from the fault injection experiment result discussed in the previous section. Therefore, we analyze them qualitatively in this section.

When a circuit enters into the wear-out phase of it's lifetime, most of the wear-out mechanisms show early symptoms as increasing signal propagation latency prior to inducing permanent device failures [24]. The ability of the HyFT and *i*HyFT architectures to detect these early symptoms and act upon by causing reconfigurations reduces the aging effects on the system by distributing the stress on two of the three CL copies. The capability of selective sparing helps reduce the rate of failures and increase the life span of circuit parts that embed such fault-tolerant architecture.

Another qualitative aspect of fault-tolerance capability is fault accumulation effect that distinguishes both considered versions of TMR from HyFT and *i*HyFT. TMR is an error masking architecture that does not indicate the presence of error, instead just corrects them until only one computational copy exhibits an error. When faults accumulate due to wear-out and multiple copies start getting affected, TMR fails to correct them and the lack of any provision of indicating error ends up in fail-silent outcomes. Whereas HyFT and *i*HyFT are able to correct errors until two faulty copies manifest the effect of fault at the output in a same way at the same time, which is very less likely. In all other possible scenarios HyFT and *i*HyFT, if cannot correct can at least indicate the presence of error and continue fail-safe operation.

VI. CONCLUSION

In order to produce a meaningful comparison of the state-of-art fault-tolerant architectures, we present herein an experimental analysis based on standard circuit analysis tools and a simulation based fault-injection framework to obtain results in terms area, power and performance overheads and fault tolerance capability. The results show that the improved Hybrid fault-tolerant architecture saves notable amount of power, while offering similar robustness improvements as

TMR. In addition it's lifetime reliability improvement and ability to deal with fault accumulation effect makes it feasible for low-power mission critical applications.

We intend to continue the analysis with larger benchmark circuits to validate the projected effectiveness of HyFT and *i*HyFT when used with larger combinational logic blocks. We also aim to perform multiple fault injection campaigns to quantitatively access the lifetime reliability improvement and the fault accumulation effects in different fault-tolerant architectures.

ACKNOWLEDGMENT

This work has been partially funded by "National Centre for Scientific Research" (CNRS) under the framework of "International Associated Laboratory - French-Italian research Laboratory on hardware-software Integrated Systems" (LIA - LAFISI).

REFERENCES

[1] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors (ITRS) 2013", Retrieved Aug, 2015 from http://www.itrs.net/reports.html2013.

[2] I. Koren, and C. Krishna, "Fault Tolerant Systems", Morgan Kauffman Publisher, 2007.

[3] R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," IBM Journal of Research and Development, Vol. 6, Issue 2, April 1962, pp. 200-209.

[4] J. Vial, A. Bosio, P. Girard, C. Landrault, S. Pravossoudovitch, and A.Virazel, "Using TMR Architectures for Yield Improvement," Int. Symp. on Defect and Fault-tolerance in VLSI Systems, Oct 2008, pp. 7-15.

[5] J. Vial, A. Virazel, A. Bosio, P. Girard, C. Landrault, and S.Pravossoudovitch, "Is TMR Suitable for Yield Improvement?," IET Computers and Digital Techniques, vol. 3, No 6, Nov 2009, pp. 581-592.

[6] M. Zhang et al., "Sequential element design with built-in soft error resilience," IEEE Transactions on Very Large Scale Integration Systems, Vol. 14, No. 12, Dec 2006, pp. 1368–1378.

[7] D. Ernst et al., "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," Proc. of the 36th Annual IEEE/ACM Int. Sym. on Microarchitecture, Dec 2003, pp. 7-18.

[8] S. Das et al., "Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance," IEEE J. of Solid-State Circuits, Vol. 44, Issue 1, Jan 2009, pp. 32-48.

[9] M. E. Imhof, and H.-J. Wunderlich, "Soft Error Correction in Embedded Storage Elements," Int. On-Line Testing Symp., July 2011, pp. 169-174.

[10] J. Yao et al., "DARA: A Low-Cost Reliable Architecture Based on Unhardened Devices and Its Case Study of Radiation Stress Test," IEEE Transactions on Nuclear Science, Dec 2012, vol. 59, no. 6, pp. 2852-2858.

[11] V. Subramanian, and A.K. Somani, "Conjoined Pipeline: Enhancing Hardware Reliability and Performance through Organized Pipeline Redundancy," 14th IEEE Pacific Rim International Symposium on Dependable Computing, Dec 2008, pp. 9-16.

[12] M. Mehrara, M. Attariyan, S. Shyam, K. Constantinides, V. Bertacco, and T. Austin, "Low-Cost Protection for SER Upsets and Silicon Defects," Design, Automation & Test in Europe Conference, April 2007, pp. 1-6.

[13] A. Benso, Alfredo, and P. Prinetto, "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation," Springer US, 2003.

[14] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, and H.-J. Wunderlich, "A Hybrid Fault Tolerant Architecture for Robustness Improvement of Digital Circuits," Asian Test Symposium, Nov 2011, pp. 136-141.

[15] I. Wali, A. Virazel, A. Bosio, L. Dilillo, and P. Girard, "An Effective Hybrid Fault-Tolerant Architecture for Pipeline Cores," IEEE European Test Symposium, May 2015, pp. 1-6.

[16] I. Wali, A. Virazel, A. Bosio, P. Girard, and M. Sonza Reorda, "Design Space Exploration and Optimization of a Hybrid Fault-Tolerant Architecture," to appear in Proc. of IEEE Int. On-Line Test Symp., 2015.

[17] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson "On Latching Probability of Particle Induced Transients in Combinational Networks," Symp. on Fault-Tolerant Computing, June 1994, pp. 340–349.

[18] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," Int. Conf. on Dependable Systems and Networks, June 2002, pp. 389-398.

[19] J. Velamala, R. LiVolsi, M. Torres, and C. Yu, "Design sensitivity of Single Event Transients in scaled logic circuits," 48th Design Automation Conference, June 2011, pp. 694-699.

[20] P. K. Lala, "Self-Checking and Fault-Tolerant Digital Design", Morgan Kauffman Publisher, 2000.

[21] D. A. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, A. Todri, M.E. Imhof, and H.-J. Wunderlich, "A Pseudo-Dynamic Comparator for Error Detection in Fault Tolerant Architectures," VLSI Test Symposium, April 2012, pp. 50-55.

[22] NanGate FreePDK45 Open Cell Library, Retrieved Aug, 2015, from: http://www.nangate.com/?page_id=2325.

[23] G. Wirth, Kastensmidt, L. Fernanda, and I. Ribeiro, "Single Event Transients in Logic Circuits_Load and Propagation Induced Pulse Broadening," IEEE Transactions on Nuclear Science, Dec 2008, vol. 55, no. 6, pp. 2928-2935.

[24] J. A. Blome, S. Feng, S. Gupta, and S. Mahlke, "Online timing analysis for wearout detection," In Proc. of the 2nd Workshop on Ar- chitectural Reliability, Dec 2006, pp. 51-60.

Mobile Application Validation through Virtualization

Cyril Dumont[1] Steven Enten[2] Fabrice Mourlin[2] Laurent Nel[1]

(1) Research department
Leuville Objects
Versailles, France
email : {cyril.dumont, laurent.nel}@leuville.com

(2) LACL
UPEC University
Créteil, France
email : {steven.enten, fabrice.mourlin}@u-pec.fr

***Abstract*—When several business applications use low level libraries which are not installed in the firmware of the device; a solution consists in building of a new firmware with the well-chosen libraries. This afford to deploy only once for all the business mobile applications. The complete adaptation of an operating system to replace the Read-Only Memories (currently called ROM) manufacturer, fundamentally changes the kernel of an embedded system. This solution allows us to offer regular and frequent custom firmware updates that maintain business applications dedicated stability in time. By creation of firmware, we leave out the space consuming trial software. We may also leave out many of the included utilities, letting our users add them back only if they need them. Often we also strip out carrier specific versions of the launcher, replacing them with Google's original versions or a version we prefer. After customizing a firmware, we focus on adding business software in place to monitor the embedded device. Thru, we use this firmware to virtualize an embedded device. Thus, we collect information to determine whether the firmware can be deployed on devices. The collected data are about memory usages, threads, and resource access and energy consumption. So, this reporting step sums up the validation of our firmware, then they are validated to a deployment step on mobile devices. Reports are delivered about the behaviors of embedded software.**

Keywords-embedded device; firmware custom; monitoring; virtualization; state management.

I. INTRODUCTION

To cook a ROM (Read Only Memory) is the process of modifying a firmware of an embedded device. It can be seen as a kind of bridge between the applications and the actual hardware of a device. When business applications need low level libraries, the firmware has to be customized by the company. For companies which need specifically designed terminals to one or more trades, rhis new solution is called the Read-Only Memories (currently called ROM) cooking. Such approach is also useful when business applications have to be added into a new kind of embedded device. Starting from a base of operating system (such as Android) installed on a smartphone or tablet, firmware, or operating system is modified to fully meet demands without unnecessary applications. Another motivation of firmware update occurred when low level libraries have to be changed. There are plenty of examples in companies: for instance a TV application needs a specific library for video streaming; a network application which encrypts its messages needs also the use of a specific algorithm which is not necessary in a standard distribution of the framework.

The firmware cooking, namely the complete adaptation of an operating system to replace the firmware manufacturer, fundamentally changes the kernel, the Android framework and pre-installed applications for a completely clean system to the company. This solution has the merit of offering and customized the firmware of regular updates and more frequent than the manufacturer's updates. This solution provides better stability over time of business applications. Android firmware is particularly suitable for customization for several reasons. Embedded device manufacturers have a process to build their own firmware. It is time consuming and the update management is a difficulty regarding the set of potential devices.

As an open system, developers have the source code required to modify the Linux kernel. They rely for the rest on the binary components manufacturer if their source codes have not been distributed. These codes are commonly referred to as hardware abstraction modules, for example for the camera, Global Positioning System (GPS), sound and graphics acceleration. Android is supported by a broad spectrum of terminals, allowing a company to choose the device ergonomics best suited to business constraints.

Finally, users are not disturbed because the Android environment is familiar. A consumer product can perfectly meet a Business to Business (B2B) demand. Experience has shown that the porting of applications is now done without difficulty, regardless of the changes made by Google's Android versions, even a major evolution [1].

Some companies that have experience in customizing firmware chose to integrate their approach, the staff concerned. The selected spectrum of users participating in the experiment, allowing finely define requirements and content and gradually reflect on future developments as a dedicated applications market. These companies are also finding true motivation of their staff on these issues. Today, in terms of support, initiatives are numerous: note of Android Business Group, the sharing of experiences between large accounts [2], the Data Android User Group [3], a monthly meeting of developers, backed by Google and able to meet many contributors. This contributes to the development of firmware cooking into several companies.

The firmware cooking allows access to personal data essential to their work wherever the end users are and the company with materials and fully dedicated to their activities such as information environments, training, home automation, transportation, geolocation, security, etc.). When new software is installed, the consequences on these data are essential. Also, what are the disruptions caused by new software on the host platform. In order to do this study, it is important to be able to place a virtual machine under observation. The goal is to validate by suitable tests that use this virtual machine are safe. These observations are crucial because the next step is the deployment of virtual machine on mobile devices. And any error becomes serious consequences for the users and the publishers [4].

We have structured our paper to present our approach to firmware customization and also put in our firmware monitoring controllers. So, the section 2 deals with the monitoring of mobile applications, the nature of the data collected but also how to make the collections to minimize disruption to ongoing observations. In section 3, we discuss a case of firmware manufacturing dedicated to the study of a business application. We add measurement points dedicated to the type of mobile platform. Then, we perform the data collection and build the associated representations. In section 4, we explain the usefulness of such software architecture for gathering information. Data analysis is also detailed. Finally, we conclude on the implementation of our approach for customizing firmware before deployment.

II. MOBILE APPLICATION MONITORING

Because new applications can involve conflicts between the previously installed applications, it is essential to observe the behavior of the mobile applications on a given device. This is particularly crucial with applications which need root permission or acquire some sensors such as a camera, etc. Mobile monitoring is become a key step in the lifecycle of a new mobile application. This step consists of looking at the behavior of an application on an embedded platform.

A. Embedded system monitoring

1) Basic mobile application monitoring:

Some fraudulent mobile applications are malware, which may capture personal information sent and received by the device or make phantom calls to premium phone numbers, while others may just be using a company name or logo without prior authorization. Regardless of their intent, these applications create a negative association in the mind of the user, which tarnishes the company's good reputation. This type of bad behavior can be detected by a sufficiently long period of observation commissioning of future embedded platforms.

When a fraudulent application is detected, it has to be immediately reported to a log along with a full report [5]. This contains developer information, number of downloads, application screenshots, and a diagnostic as to why this mobile application is believed it to be fraudulent. It provides valuable intelligence data and can help support a criminal investigation.

Today, manufacturer services give security operations users an additional layer of protection, coupled with a new data stream that includes more contextual information about the specific and potential threats contained in fraudulent mobile applications. This approach can be completed by ad hoc tools, which collect data about runtime of applications under observation. We have decided to build a tool chain for building these data collections.

2) Adhoc monitoring.

Through the application monitoring feature a mobile application can be studied in depth if the monitoring task is developed in close relationship with the features of the given mobile application. For instance, when a mobile application uses Bluetooth protocol, then a monitoring task has to be configured to control the packets, which are transferred on this protocol, the collisions which occur, the availability of the sensor, etc.

Tools such as Systrace tool, helps us to analyze the performance of mobile applications by capturing and displaying execution times of these applications processes and other Android system processes [6]. This kind of tools combines data from the Android kernel, such as the Central Processing Unit (CPU) scheduler, disk activity, and application threads to generate an Hypertext Markup Language (HTML) report that shows an overall picture of an Android device's system processes for a given period of time. Very often, such kind of tools is considered as debugging tools because they are particularly useful in diagnosing display problems where an application is slow to draw or stutters while displaying motion or animation. But a main drawback is the obligatory use of a USB debugging connection.

We needed a way to get periodic screenshots of a mobile device connected to a computer through a light protocol. On Android Platform Dalvik Debug Monitor Server (DDMS) has the ability to take screenshots on-demand, but not automatically. It provides port forwarding services, screen capture on the device, thread and heap information on the device, etc. but the documentation is so poor that source code of the library is the only information source. It uses an Android Debug Bridge called `adb`. It allows us to communicate with a connected device on the same WIFI network.

On Android, every application run in their own process, each of which runs in its own Virtual Machine (VM). Each VM exposes a unique port that a debugger can attach to. We have built a DDMS monitoring application for looking at the embedded runtime of business applications. When we start our DDMS application, it connects to `adb`. When a device is connected, a VM monitoring service is created between `adb` and our DDMS monitoring application, which notifies our DDMS application when a VM on the device is started or terminated. Once a VM is running, our DDMS application retrieves the VM's process ID (pid), via `adb`, and opens a connection to the VM, through the `adb` daemon (`adbd`) on the device. Our DDMS application can then talk to the VM using a custom wire protocol. The result is a data collection about the behavior of the embedded business application.

This strategy can be translated within a hypervisor like VirtualBox or VMWare. The `adb` daemon is called through a virtual network mapping between the host platform and the

Android virtual machine. The main advantage of this approach is to run the monitor on the host platform and the mobile application (under observation) on a virtual machine managed by a hypervisor. A second benefit is on the porting of application. The hardware architecture constraints are respected; only the configuration of our monitoring application is updated and its network mapping.

B. *Memory management*

Our DDMS application allows us to view how much heap memory a process is using. This information is useful in tracking heap usage at a certain point of time during the execution of business applications. Another feature of our DDMS application is to track objects that are being allocated to memory and to see which classes and threads are allocating the objects. This allows us to track, in real time, where objects are being allocated when we perform several actions in our application. This information is valuable for assessing memory usage that can affect application performance. This happens when an application shares preferences with another one.

The file system of the virtual machine is also an information source. It is useful in looking at files that are created by a mobile application or if we want to transfer files to and from the virtual machine. This is also useful when the size of the data collection is so large that it is suitable to filter a part of the data before the transfer. This case occurs when the mobile application uses the sensors such as the camera or the microphone. The output format and the recording involve often a large output file. Only a part of the data is useful for the analysis. Also, we filter locally to the device a subset of persistent data by the end of the monitoring scenario.

C. *Time profiling*

Method profiling is a means to track certain metrics about a method in a program, such as number of calls, execution time, and time spent executing the method. When we want more granular control over where profiling data is collected, it is possible to deep into the body of a method and to compute other measures.

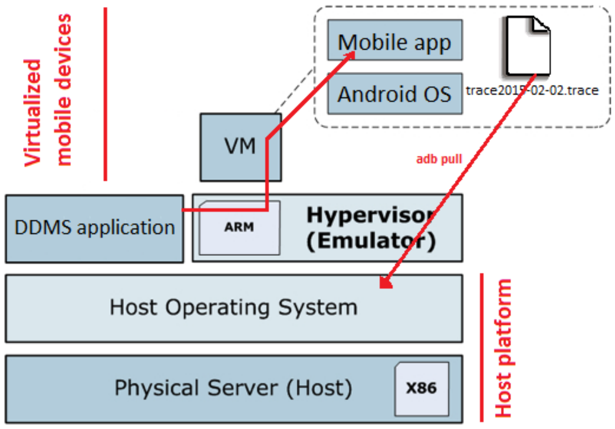


Figure 1. Monitoring architecture

A difficulty of embedded operating systems such as Android lies in its organizational changes between different versions of the same operating system. So, depending on the Android version, our DDMS application provides a summary of what happened inside a given method. Also, we need to generate log files containing the trace information we want to analyze. We use the `Debug` class in our code and call its methods such as `startMethodTracing()` and `stopMethodTracing()`, to start and stop logging of trace information to disk. This option is very precise because we can specify exactly where to start and stop logging trace data in our DDMS application. Our monitoring application has necessary the permission to write to external storage.

To create the trace files, we include the `Debug` class and we call one of the `startMethodTracing()` methods. In the call, we specify a base name for the trace files that the system generates. These methods start and stop method tracing across the entire virtual machine. For example, we could call `startMethodTracing()` in our activity's `onCreate()` method, and by the end of the monitoring stage, we call `stopMethodTracing()` in that activity's `onDestroy()` method. When our application calls `startMethodTracing()`, the system creates a file called "`trace2015-02-02`" trace. This contains the full method trace data and a mapping table with thread and method names (see figure 1).

The system then begins buffering the generated trace data, until our application calls `stopMethodTracing()`, at which time it writes the buffered data to the output file. If the system reaches the maximum buffer size before we call `stopMethodTracing()`, the system stops tracing and sends a notification to the console. This event can also trigger the pulling of the technical data from the mobile device to the workstation. We have also used the data exportation through the use of RESTful remote monitoring application.

D. *Profiling scenario*

After a mobile application has run and our DDMS application has created the trace files "`traceyyyy-MM-dd`".trace on the device, we have to copy those files to the host computer. We use `adb pull` to copy the files. As an example, we copy a trace file, `trace2015-02-02`, from the default location on the device to the `/tmp` directory on the host machine via:

```
   adb pull /sdcard/trace2015-02-02 /tmp
```

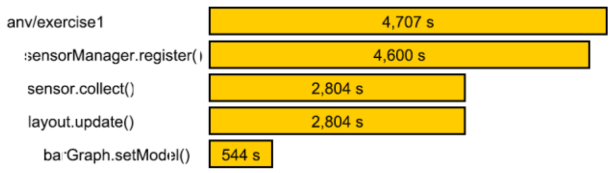


Figure 2. top 5 of costly methods

The format of the trace file is suitable to be parsed by `TraceView` or `TraceDump`. This tool uses the `Graphviz Dot` utility to create the graphical output, so we need to install `Graphviz` before running this command. But as we are going to explain in the fourth section, the data can also be sent to a monitoring server where services parse the collected data and compute metrics about the execution [7]. As an example, the Figure 2 shows the most costly methods into a bar graph representation.

III. CUSTOM STRATEGY FOR FIRMWARE COOKING

Firmware is the low level programming. They are also often called firmware; they contain the operating system and basic applications to make the phone work. For the iPhone and iPad those firmware come from Apple and can typically only be updated when Apple issues updates. But for Android devices there are literally hundreds of developers working on custom firmware for most common models of phones and tablets, which they are proud to share with the community. In our working context, we need to define firmware with additional software. First, we need to add our DDMS application for the future observations. This component is important for local monitoring. Secondly, we want to install business software which is under control during this validation step.

A. From source to Virtual Machine

For building a new firmware, several choices have to be done. So, it is essential to know the advantages or disadvantages of using an Android Open Source Project (AOSP) firmware versus a ROM stock. A ROM stock is the firmware that comes with a device; the device is stocked with that firmware by the manufacturer. Android is generally customized by the manufacturer to some degree; at minimum there needs to be device specific drivers for Android to work on a particular device. The customizations may include a custom theme, launcher, and default applications like monitoring control panel does.

1) A large set of acronyms

An AOSP firmware is a ROM based on the Android Open Source Project. In the purest sense, AOSP refers to unmodified ROMs or code from Google. The name is often co-opted for a custom firmware that is very close to the original AOSP, since these firmware still need to be customized; for example, we have downloaded and compiled the Android source code and run it on a Samsung Galaxy S5 with doing a whole lot of customizations. For example, monitoring libraries are installed with test suites. This means that we have added source projects with configuration files for building, testing.

Technically, ROM stocks are all AOSP firmware apart from the versions of Android that has not been released yet. Kitkat and Lollipop firmware are AOSP for a long time; the source code is available at Android Web site. In the next case studies, we will use Kitkat and Lollipop versions.

To further add to the confusion, a custom firmware does not refer to customized firmware in general. That term specifically refers to firmware that has been customized by engineers or researchers which are not the manufacturers or carriers. For example, CyanogenMod provides firmware [8] which is modified under the constraint of the open source community. Most AOSP firmware for a specific device is ROM stocks that have been customized to remove some of the manufacturer or carrier features and make them closer to the pure AOSP experience. As an example, we can disable PIE feature on any ROM stock or AOSP firmware. The option is not even available in most ROM stocks. So our solution is to modify the AOSP firmware source and then build them into an updated firmware.

2) The benefits of firmware cooking:

First of all, the main thing to know is that messing with the firmware of phone can be risky. We can potentially damage a mobile phone so that it won't be usable without some major low-level hacking. This reason involves our need to experiment new customized firmware behind a hypervisor.

The most basic benefit of custom firmware is getting rid of unstable software of malware, spy application and so on. These applications take up precious room on a mobile device. Beyond simple fixes, custom firmware can also open a whole world of new possibilities for new mobile devices. In many cases newer versions of Android are available for the devices as custom firmware, beyond what the carrier has released or is planning to release. Custom firmware can also include other pleasant features, like overclocking, themes, private browsing support, and so on.

Our current objective was to isolate the minimum Android operating system which can support our business mobile applications and our monitoring tools. The architecture of Android platform is defined to host new software. The source codes provided by AOSP are also organized to host new source projects, which have a predefined structure. Android has layer architecture (Figure 3). This structure is understandable easily when we explore the source code. For instance, we wanted to record, convert and stream audio and video. We observed that, the OpenGL library can be upgraded or completed by the add-on of `ffmpeg` library. Also, we have added the source code project with Android build file into the whole Android source files. When a new build is launched, then this new library is taken into account.

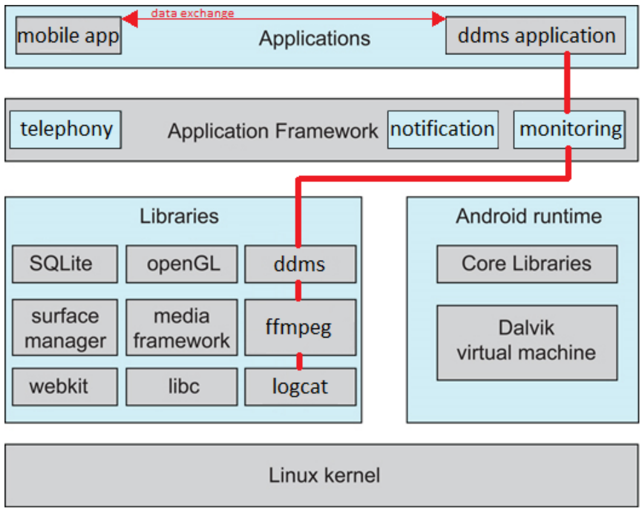


Figure 3. Android layer architecture

For our project, the main benefit was the enrichment of the Android source distribution with the source codes of all the boxes connected with the thick red line (Figure 3). The box called "mobile app" plays the role of the test suites or the mobile application we want to observe before validation.

Next, a benchmark step can start. Its duration depends on the features, we want to observe. Software stability is a key feature in the study of our business applications. Also, the meantime declared by the manufacturers are used as test limits

3) Our firmware cooking approach:

A firmware construction is a step by step process. First, we define the software basement. Then, we prepare the build of a new firmware, with the selection of the right add-ons. Controls are done about the version numbers, compatibility features, architecture definition and security permissions. We have developed an embedded monitoring application for our embedded devices that generates log files with technical data about the runtime. In the future, new tracking tools will be added about energy management, bandwidth network use. So, if we imagine those applications always having to write the machine code to get the monitoring to turn on. It would be a lot of code duplication and would make an application slow. Instead, for functions like the method profiler or thread manager, we have packaged our own monitoring library. These are pieces of codes that can be executed by calling them through a method call. These are already pre-written and ready to use. It saves a lot of coding work and keeps the source code small. On the Android device, we have particular libraries like `ddms` or `ffmpeg` libraries that can't be absent, or else the firmware won't even be used in our studies.

On Figure 3, the red line highlights the dependencies during a monitoring scenario. A mobile application is under observation. Also, by the use of our monitoring application (called `ddms` application), we collect behavioral data. They are computed through the use of `ddms` library and `logcat` library which already belongs to AOSP distribution. Depending on the application domain of the mobile application, the `ffmpeg` library plays a role of stream observer. For instance, it contains `ffprobe` tool which gathers information from multimedia streams and prints it in human- and machine-readable fashion. For example, we use it to check the format of the container used by a multimedia stream and the format and type of each media stream contained in it. We use `ffprobe` in combination with a textual filter, which performs more sophisticated processing, e.g., statistical processing and plotting. `ffprobe` output is easily parsable by a textual filter, and consists of one or more sections of a form defined by the selected writer, which is specified by the print format option. The sections contain other nested sections, and are identified by a unique name. The metadata tags stored in the container or in the streams are recognized and printed in the corresponding output section. This means that we control the properties of the video streams. Such abilities are essential in the case of audio communication application, such as encoder/decoder audio applications. Depending on the data size, the output files are saved into a specific folder. Each of them respects a block size limit.

B. Virtual Machines managed by hypervisor

The virtualization is a technology that allows resources to be shared by a variety of physical outlets. Today, virtualization is a topic, which is focused on the relatively new concept of server virtualization. In this context, multiple operating system and application sets are virtualized on a single server, allowing it to be more efficiently and cost effectively used. But, there are a multitude of virtualization schemes addressing a spectrum of applications. We have addressed new ideas around virtualization and applied their uses and advantages to the virtualization of mobile devices.

1) Virtualization of platform.

The virtualization is an abstraction over physical resources to make them shareable by a number of physical users. Platform virtualization is what enables both server and desktop virtualization. A platform in this context refers to the hardware platform and its various components. This includes not only the CPU, but also networking, storage and bus attachments such as USB and serial ports, but also sensor such as camera, microphone and even GPS or compass.

The key technology that makes this possible is called the hypervisor. The hypervisor is the component that virtualizes the platform, making the underlying physical resources shareable and implementing the policies for sharing among the multiple virtual machines. These ones can belong to embedded systems like Android or IOS. They are an aggregation of the operating system and application set which contains our mobile applications. The VMs are considered as a file in some format depending of the hypervisor. The virtual disk used by the VM is another file encapsulated within the VM.

An Android VM as a file in a host system like a hypervisor has some interesting benefits: we can back up the full virtual machine and its configuration in one hit rather than backing up at file level within the server. As a file, it's easy to manage a VM as a template. It's also simple to move a VM from one host system to another, as the process is nothing more than a file copy. As we expected with the virtualization concept, there are a variety of ways in which virtualization can be achieved. For platform virtualization, there are two primary models, called full virtualization and para-virtualization. Both are suitable depending on the kind of architecture of the mobile device ARM or x86. The last case corresponds to full virtualization; the former one exploits an intermediate layer.

2) Managed VM through a hypervisor .

The hardware of mobile devices has multiple features; this concerns not only the processor but also the sensors and even the pluggable add-ons. The usage of VM involves several approaches depending on the architecture of the physical server. In our project, we use AMD architecture servers (64 bits). This means that we apply the full virtualization when we test and validate software for AMD mobile devices. This kind of virtualization provides a sufficient emulation of the underlying platform that a guest operating system and application set can run unmodified and unaware that their platform is being virtualized. But, the most widespread kind of processor is ARM, a solution is to make the guest operating system aware that it's being virtualized. With this knowledge, the guest operating system can short circuit its drivers to minimize the overhead of communicating with physical devices.

We use Qemu, which emulates a full system, including a processor and various peripherals [9]. A number of specific emulator features are enabled in both the Android kernel and Android user space environment when run in an emulated environment. These features allow a smooth and complete user experience resembling using a real Android device, on laptop and desktop workstations. With the introduction of the ARMv8-A architecture and Android support for 64-bit ARM platforms, this need is more important than ever because it allows us to begin adapting our applications to an ARM 64-bit based mobile ecosystem prior to hardware being available. All of our tests are based on the use of Qemu and the exposition of our own custom firmware via a graphical interface.

C. *Testbed of mobile business applications*

Testing functionality is typically a matter of enumerating the functions that an application should support, then defining a set of tests that exercise those functions, with pass/fail results. Problems that we encounter running functional tests are input to evaluating usability. For usability, we want to have several mobile end users with different skill levels attempting to accomplish a given set of business goals, producing subjective ratings that indicate how easy or hard the task was. Performance test results are easier to quantify, but can be very difficult to interpret. For example, wireless throughput is always higher in the lab under ideal conditions than in real life, so be very careful about the conclusions we draw from performance tests.

To test failure modes in components, we enumerate a number of possible failure conditions and simulate them. We must also identify what we are measuring. For example, when measuring time to establish the connection in the event of network loss of signal, do us measure network connection resumption or mobile application connection resumption.

IV. Custom Approach of Mobile Application Monitoring

Based on the build of our own kernel and the enrichment of the AOSP sourced, we have built our own custom firmware. Our results are presented as log reports and numerical measures.

A. *Monitoring architecture for mobile applications*

When a mobile business application is running under monitoring, all events are saved through the use of a local monitoring application (as explained previously). Huge amount of data, even for relatively small programs are recorded in the local file system. Then, these data are exported to a server. The main events are class load, or unload, compiled method load, and unload, GC start, finish, method entry and exit, thread start and end, etc. We assign IDs to objects, classes, methods, etc. And our monitoring application is responsible for keeping track of IDs. They are assigned through defining events (e.g., class load). As a small part of an example of output trace, the following sequence of event trace in table I.

TABLE I. Event Trace of Method callIntent

```
public int callIntent(int);
46: iload_1
47: iconst_2
48: irem
49: iconst_1
50: if_icmpne 54
51: iconst_2
52: istore_2
53: goto 56
54: iconst_5
55: istore_2
56: iload_2
57: ireturn
```

Other information about performance is also collected. They are about the method execution measure and also class loading and checking. As an example, the table II shows a first level of information about time measures. The size of these data depends on the number of samples per time unit.

TABLE II. Data Time Collection

```
Capture.callIntent 2015-03-08 14:59:30.252,
Capture.update 2015-03-08 14:59:30.254,
Model.get 2015-03-08 14:59:30.255,
Capture.setProps 2015-03-08 14:59:30.258 …
```

All the timestamps allow tester to display the events of the garbage collector at runtime.

B. *Interaction between monitor and mobile application*

Our message exchange protocol is packet based and is not stateful. There are two basic packet types: command packets and reply packets. Command packets may be sent by either the ddms application or the target VM. They are used by the ddms application to request information from the target VM, or to control program execution. Command packets are sent by the target VM to notify the ddms application of some event in the target VM such as a breakpoint or exception. A reply packet is sent only in response to a command packet and always provides information success or failure of the command. Reply packets may also carry data requested in the command (for example, the value of a field or property). Currently, events sent from the target VM do not require a response packet from the ddms application.

Our monitoring protocol is asynchronous; multiple command packets may be sent before the first reply packet is received. The layout of each packet looks like in table III:

TABLE III. Command Packet layout

```
Header
   length (4 bytes)
   id (4 bytes)
   flags (1 byte)
   command set (1 byte)
   command (1 byte)
```

```
data (Variable)
```

All fields and data sent via our monitoring protocol should be in big-endian format. It means the big-end is first. In other words, we store the most significant byte in the smallest address

TABLE IV. REPLY PACKET LAYOUT

```
Header
   length (4 bytes)
   id (4 bytes)
   flags (1 byte)
   error code (2 bytes)
data (Variable)
```

The length field is the size, in bytes, of the entire packet, including the length field in table IV. The id field is used to uniquely identify each packet command/reply pair. Flags are used to alter how any command is queued and processed and to tag command packets that originate from the target VM. The command set is useful as a means for grouping commands in a meaningful way. The error code field is used to indicate if the command packet that is being replied too was successfully processed.

This command field identifies a particular command in a command set. This field, together with the command set field, is used to indicate how the command packet should be processed. The data field of a command or reply packet is an abstraction of a group of multiple fields that define the command or reply data. As an example of data type: threadID uniquely identifies an object in the target VM that is known to be a thread. Another example is methodID, which must uniquely identify the method within its class/interface or any of its subclasses. A methodID is not necessarily unique on its own; it is always paired with a referenceTypeID to uniquely identify one method. The referenceTypeID can identify either the declaring type of the method or a subtype.

C. Validation process of mobile application

The list of all the data types is not exhaustive here but all element of a runtime program can be referenced and tracked. So, based on these results, we are able to decide whether the source codes of mobile applications can be added into the source.

The validation process takes into account response time of applications and our monitoring protocol helps us to collect internal data from each application under test. For instance, when response time is greater than 20% of the expected response time, we can conclude that there are perturbations from the runtime context towards the application sunder test.

Another test case is about the management of the memory by the virtual machine which runs a business application. We can compare the used memory with the first benchmarks of the applications under test. When the difference exceeds 25% then if means that the application cannot be deployed on the future firmware.

If we do not respect such rules, we could build unstable firmware and the consequences will be more serious. For instance, after flashing the firmware a phone works fine for one or two weeks. But soon this phone starts crashing more often. The phone starts rebooting every now and then it becomes useless. The most difficult point is the loss of working time. Also, by applying the reference measurement definition tested is crucial for our validation process

The validation by the use of virtualization has the advantage of using virtual devices instead of concrete smart device. The flash of firmware is a dangerous operation for the hardware and we preserve the hardware by previously testing our custom firmware. Another approach need a deployment on a smart device with a rooted firmware.

In the opposite side, the build of firmware involves new drawbacks after the deployment step. A first one is a legal issue. This means that the manufacturer guarantee is cancelled when a free firmware is installed. A second one is about the telecom provider checks. When several tools of a given telecom provider are already installed, then exceptions are raised by these applications when the underlying firmware is changed. Also, it is often useful to change a whole toolkit of software when new firmware is built by ourselves.

V. CONCLUSION

As we explained in the first section, we need to prepare our own firmware because of the change of libraries which are essential for our business applications. Also, the choice of validation before deployment explains our use of virtual machine. In this paper, we have presented our approach of the monitoring of mobile business applications. It is based on a perfect configuration of all the ROM stock and its build. We have shown that it is preferable to have a local monitoring instead of a remote monitoring application. The impact of its actions is less in the case of embedded systems.

We have described briefly our stateless protocol between the VM of the business application and our monitoring embedded application. We want to enrich this protocol and then observe new kinds of property.

The data collected are exported to a server where they can be parsed and aggregated with other simulations. Next new reports can be built and published onto a Web server if the monitoring data are public. We think that our approach is an adaptation of a monitoring strategy from Web domain into the domain of mobile applications. We consider our pragmatic study as a validation of our concepts and our next step will be to automatize as much as possible all the steps described in the document. And so, our experience could be transferred to other development teams.

Our approach reduces the effort of deployment on a large set of devices. Moreover, we reduce also the number of anomalies by increasing the observation time when some expected benchmarks are not achieved

REFERENCES

[1] "The Android Source Code: Governance Philosophy", source.android.com, January 25, 2015.

[2] M. Isacc, “A deep-dive tour of Ice Cream Sandwich with Android's chief engineer”, Ars Technica, September 15, 2012.

[3] A. Shah, “Google's Android 4.0 ported to x86 processors”, Computerworld, International Data Group, February 20, 2012.

[4] R. Whitwam, “HTC Posts Android 4.4 Kernel Source And Framework Files For One Google Play Edition, OTA Update Can't Be Far Off”, androidpolice.com, December 2, 2013.

[5] “Exclusive: Inside Android 4.2's powerful new security system | Computerworld Blogs”, Blogs.computerworld.com, November 9, 2012.

[6] “AppAnalysis.org: Real Time Privacy Monitoring on Smartphones”, February 21, 2012.

[7] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo, “A technique for drawing directed graphs”, IEEE-TSE, March 1993.

[8] E. Tyler and W. Verduzco, “XDA Developers: Android Hacker's Toolkit: The Complete Guide to Rooting, ROMs and Theming”, Wiley edition, May 2012.

[9] R. Warnke and T. Ritzau, “Qemu”, Paperback, March 10, 2009.

Variability in Test Systems: Review and Challenges

Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria

Computer and Electronics Department
Mondragon Goi Eskola Politeknikoa
Goiru 2, Arrasate-Mondragon, Spain
Email: {aarrieta, gsagardui, letxeberria}@mondragon.edu

Abstract—**Customizable products are increasing in our society, which creates a need on software systems, embedded systems and cyber-physical systems to handle variability. In addition, many companies are moving towards continuous integration and deployment and as a result, an automated solution to testing the relevant configurations is needed. Thus, variability appears in different stages of the product life-cycle. When validating these configurable systems, the test framework has to deal with their variability in order to test different system variants. Modelling variability in the test systems is an elegant solution to test and validate variability-intensive systems. This paper studies some test systems that handle variability and compares their characteristics and limitations, which can help test engineers needing a variability handling test system to choose among different approaches.**

Keywords–Test Systems; Test Architecture; Variability; Validation

I. Introduction

Variability is the ability to change or customize a system [1], and it can be understood as configurability or modifiability [2]. In the case of configurability, the variability appears in the product space, whereas in the case of modifiability, variability appears in the time space. The discipline of representing variability in models that describe the common and variable characteristics of a product is named variability modelling [3].

The different demands of the society and the users are some of the causes for customized products. As a consequence, variability in different points of the products, systems and development phases is increasing. Variability-intensive systems can be configured into thousands or millions of system variants. From one system variant to another, variability can appear not only in the product itself, but also in the elements in charge of testing the product, i.e., test system.

When testing different system variants, the test system has to be configured and adapted as well in order to test them. Due to this issue, modelling variability in the test system can be an interesting approach for testing variability-intensive systems, such as software product lines (SPLs), configurable embedded systems or configurable Cyber-Physical Systems (CPSs).

Most of the reviews and surveys in the field of variability modelling focus on the variability of the system itself, e.g., [4], or the use of variability modelling in industrial practice, e.g., [3], where the different notations used for modelling variability are analysed. This paper presents a review of variability-handling test systems. To carry out this study, we have systematically reviewed the documented approaches.

The rest of the paper is structured as follows: A brief introduction about the background of test systems is presented in Section II. Section III explains the methodology that has been used to systematically review the documented approaches in journals and conference papers. Section IV presents the obtained results after applying the search methodology; the documented approaches are explained and a discussion and analysis is provided. Section V defines a set of open challenges about variability modelling in test systems. Finally, Section VI summarizes the obtained conclusions.

II. Test Systems

A test system is a set of components that interact with the objective of testing the System Under Test (SUT). The complexity of a test system can vary depending on the overall test objectives and type of testing. Some of the tests are performed in simulation, e.g., Model-in-the-Loop (MiL) or Software-in-the-Loop (SiL) simulation, whereas other tests are performed in emulation, where additional hardware is needed, e.g., Hardware-in-the-Loop (HiL). Other test systems support test automation, where a test scheduler that decides which test case to execute is mandatory.

The organization of the group of components comprising the test system is called the test architecture [5], which specifies the interaction among the different elements of the test system and the SUT. A test architecture is a necessary artefact in test and validation activities so that verification and validation activities can be systematic, and it allows the reuse of test cases along the different test phases [5].

Test cases are part of the test system and provide information about the test execution. In Model-Based Testing (MBT), test cases are automatically generated either from the System Model, i.e., from the model of the SUT or from a test model [6]. When the test cases are executed, the test results have to be determined. This is typically performed by other elements of the test system, such as test oracles, which are mechanisms that analyse the SUT output and are able to decide the test result [6].

Modelling variability in the elements of the test systems allow the execution of tests under different conditions. Moreover, requirements of a configurable system varies from a system variant to another, and the test system has to be adapted in order to test different system variants, thus, variability in the components of the test system can help to achieve this goal.

III. Search Method

This paper collects the results of a systematically developed state of the art study about variability-handling test systems. To carry out this study systematically, the guideline presented in [7] has been taken as a base, which follows the presented steps below:

- Definition of the research questions
- Search process
- Inclusion and exclusion criteria
- Data collection
- Data analysis

A. Definition of Research Questions

As mentioned above, the scope of this study is to analyse the current state of the art in the field of variability handling test systems. The Research Questions (RQs) to carry out the goal were the following:

- **RQ1**: Which are the approaches documented that take into account variability in test systems?
- **RQ2**: Which kind of systems are tested with the selected approaches?
- **RQ3**: Which are the used modelling or programming languages?
- **RQ4**: Which are the used test strategies?
- **RQ5**: Which is the variability modelling approach?

B. Search Process

The search process has been a manual search by using search strings on different scientific databases with the aim of identifying conference proceedings and journal papers since 2008. The used databases have been IEEE Xplore, ACM digital library, Science Direct and Springer. Other places and sources such as proceeding of VALID 2014 or relevant PhD theses available on the Internet have been also used. Once identified some conference papers, journal papers and PhD theses on these databases, some references of the selected studies have also been used to detect new papers that are not available on the proposed databases.

With respect to search strings, the following keywords were used in order to find new papers:

- (“Variability” OR “Variant” OR “Modifiability” OR “Configurable”) AND (“Test System” OR “Validation Environments”)

C. Inclusion and Exclusion Criteria

The inclusion criteria for selecting a paper was the following:

- The publication should be “journal”, “conference proceeding”, or “PhD Thesis”.
- The reader should clearly deduce that the test system or architecture handles variability.

The exclusion criteria for excluding a paper was the following:

- The publication is not written in English.

D. Data Collection

The data extracted from each selected study was:

- Full reference (authors, title, journal or conference and year)
- Institution or institutions of the authors
- Characteristics and limitations of the proposed approach

E. Data Analysis

The data analysed once extracted the needed information was the following:

- Which is the main characteristic of the approach
- Which kind of systems are tested with the proposed approach
- Modelling tools used for the development of the test system
- Which are the main limitations of the selected approaches
- Variability points in the test system
- Variability modelling approach

F. Search Result

After applying the search strings in the aforementioned scientific databases, it concluded with 87 publications in IEEE, 12 in ACM, 1 in VALID 2014 and 1 known PhD thesis. However, most of the publications did not offer the expected overview. After following the inclusion and exclusion criteria, nine publications have been selected for the review, which are explained below. The selected publications are the following (addressing RQ1): [5][8][9][10][11][12][13][14].

IV. Variability in Test Systems

Testing variability-intensive systems is a very time and resource consuming activity due to the high number of possible variants. When a specific system variant has to be tested, the test system in charge of testing it has to be configured. The variability among the different system variants affects the test system, where different test cases have to be executed to validate a specific system configuration, and as a result, variability in the test model is required [15].

Variability handling systems appear in a wide range of systems, such as SPLs, embedded software and systems, mobile applications, CPSs, etc. The variability-handling test system approaches presented in the selected publications are used to test the following kind of systems (addressing RQ2):

- Non configurable Embedded Systems: [8][9]
- Variability handling systems (e.g., SPLs, highly configurable CPSs, etc.): [11][12][13][14][5]

TABLE I. COMPARATIVE TABLE OF THE SELECTED TEST SYSTEMS

Ref.	Modelling Language	SUT	Test Strategy	Variability Points	Variability Management	Variability Modelling
[8]	Messina	Embedded Systems	Evolutionary	Evolutionary algorithm configuration variables, target configuration	Not considered	Not used
[9]	Simulink	Embedded and CPSs	Functional Testing	In test stimuli generation	Not considered	Uniform variability
[5]	Simulink	Configurable CPSs	Functional Testing	SUT, Test oracles and Test data generator	Feature Models	Variant of negative variability
[11]	UML and UTP	SPLs	Not specified	SUT, Test context, Test cases, test component, data pool, data partition and data selector	UML sequence diagrams and a proposed UTP extension	UTP extension
[12]	UML and UTP	SPLs	Not specified	SUT, SUT interface, test context, data pool, data partition	OVM, UML and UTP	UTP extension
[13]	State Machines	SPLs	Regression testing	SUT, Test model, test goals, test suite and test plan	Not considered	Delta modelling
[14]	State Machines	SPLs	Incremental testing	SUT, Requirements, test model, test goals, test suite and test plan	Not considered	Delta modelling
[10]	Home-grown	General purpose software	Not specified	User interface, test control, code generator, information system and gateway	Not considered	Not used

- General Purpose Software: [10]

In addition, there are different tools, modelling or programming languages for developing these kind of systems. As a consequence, the modelling languages used for developing variability-handling test systems differ (addressing RQ3):

- MATLAB/Simulink: [9][5]
- State machines combined with delta modelling: [13] [14]
- UML-UTP: [11][12]
- Messina: [8]
- Home-grown: [10]

Other characteristics of test systems are the test strategies used to test the SUT. In the selected approaches, we have detected different test strategies (addressing RQ4): In [8] evolutionary testing is used, in [9] and [5] functional testing, in [13] regression testing and in [14] incremental testing. Other approaches do not consider the test strategy or it is not clear for the reader, e.g., [10].

With regard to variability modelling (addressing RQ5), different approaches are considered: Zander-Nowicka uses uniform variability modelling in [9]. Lity et al. [13] and Dukaczewski et al. [14] use delta modelling. Perez et al. use UML and UTP extension to model variability in [11] and [12]. In our previous work, we generate a skeleton model that acts as a core model, and when configuring a specific variant, the selected components of the skeleton model are replaced by models stored in a Simulink library, and the non selected are removed [5]. Other approaches ([10][8]) do not model variability. Table I summarizes the main characteristics of each approach.

A. Variability Handling Test Systems

The approach presented in [8] shows an evolutionary test system, primarily based on the MESSINA tool, that tests functional and non-functional properties of embedded systems. An evolutionary algorithm is an optimization technique based on the principles of the Darwinian theory of evolution, where a set of candidate solutions called individuals are selected. The fitness of these individuals are evaluated by the evolutionary algorithm by executing a problem-specific fitness function. The proposed approach by Kruse et al. in [8] supports MiL, SiL, Processor-in-the-Loop (PiL) or Hardware-in-the-Loop (HiL) test platforms, and allows the reuse of test cases across them. In the case of MiL and SiL test system configurations, MESSINA supports different tools, e.g., MATLAB/Simulink, ASCET models, etc. In the case of HiL, MESSINA is connected to modularHiL, a universal HiL test system developed by Berner & Mattner. The main variability points of this approach can be found in the configuration variables for the evolutionary algorithm, (e.g., mutation rate, crossover rate, etc.) as well as the test system target configuration, i.e., MiL, SiL, PiL or HiL.

Model-in-the-Loop for Embedded Systems Test (MiLEST) is a toolbox for MATLAB/ Simulink developed by Zander-Nowicka in [9]. This test system is designed towards the validation of automotive real-time embedded systems in Model-in-the-Loop (MiL). The hierarchy of MiLEST is divided into four abstraction levels: Test Harness level, Test Requirement level, Test Case level and Feature level. Although the approach in [9] proposes mechanisms for modelling variants, as shown in Figure 1, the test system itself is not designed for the validation of variability-handling systems. The proposed modelling technique is uniform variability. This variability modelling technique allocates all the components in the modelling framework, i.e., Simulink, and the variability is bound with different mechanisms, e.g., switch and constants. As there are unused components allocated in the simulation framework while simulation is running, simulation time is increased as explained in [16].

Our previous work [5] presents a configurable test architecture for the automatic validation of variability-intensive CPSs, together with a model-based process (Figure 2) for the systematic validation of these kind of systems. Variability of the test system is managed using the tool FeatureIDE [17] and saved into a *.xml file. This file is read by a test architecture generator that semi-automatically generates the skeleton of the test system. The tool FeatureIDE also allows generating different product configurations either automatically

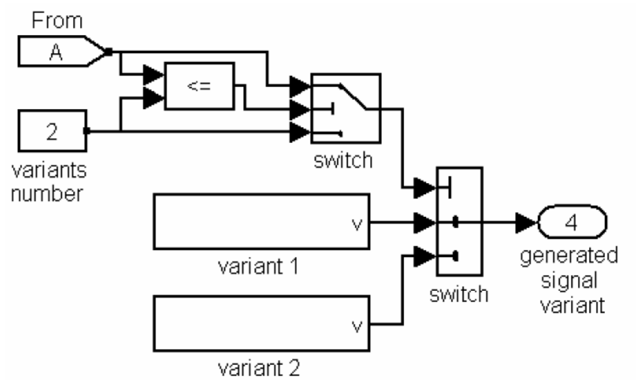


Figure 1. Variability Modelling Mechanism of MiLEST [9]

(using pair-wise or t-wise techniques) or manually. These configurations are saved into a *.config file, which is read by the test configurator. The test configurator automatically configures the test system for the selected system configuration. Finally, the work describes the different variability points of the components of the test architecture: variability of the SUT, variability of the test data generator, variability of the test oracles and test control.

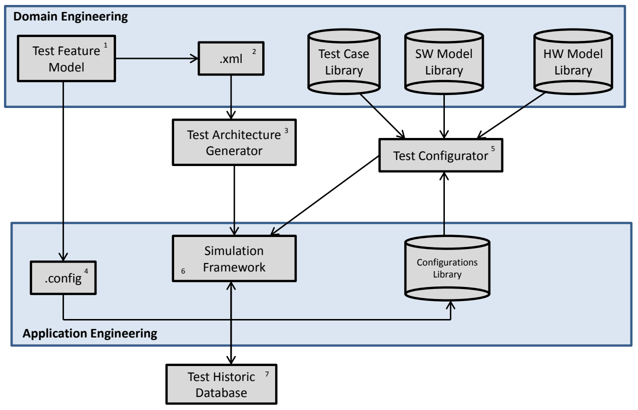


Figure 2. Model-Based Testing process for the systematic validation of variability-intensive CPSs [5]

A product line of validation environments with variability to test different applications in different domains and technologies is proposed in [10]. The study presents a validation environment able to test different SUTs from different domains, used programming languages, etc. Different elements of the validation system are identified (Figure 3) and the variability points together with variability requirements are identified and classified in a table.

The validation system proposed by [10] works as follows: The test engineer executes a test through the GUI, the GUI sends the test command to the engine, and this transforms the test command into the programming language that the SUT understands. For this step, the engine communicates with the database to obtain the correspondences between the source and target languages. When the transformation is finished, the command is sent to the SUT through the SUT interface, and awaits the response to begin the process again. These steps are shown in a UML sequence diagram depicted in Figure 4. The variability points of this system includes the user interface, test control, code generator, information system or gateway.

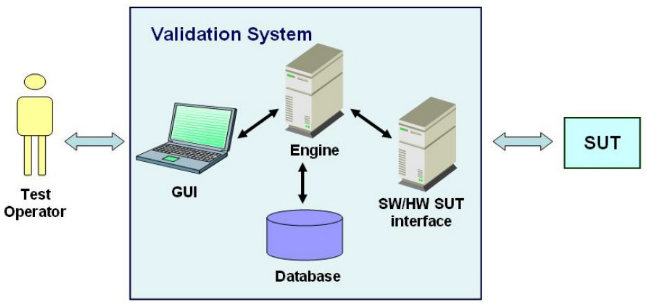


Figure 3. Test Elements Composing the Validation System [10]

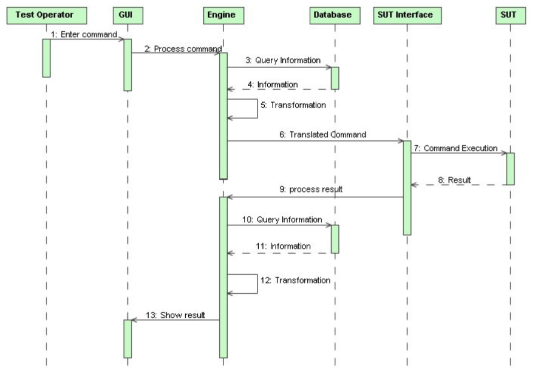


Figure 4. Interactions between the elements of the validation system proposed in [10]

The study presented in [11] defines an extended architecture for UTP to deal with variability in the test models, where the meta-model is shown in Figure 5. The proposed extension includes mechanisms to describe the behaviour of test cases and other elements needed to support variability. An example is illustrated in Figure 6.

The main variability in the proposed UTP extension is included in the Test Context, Test Cases, Test Components, UTP and Data Pool, Data Partition and Data Selector:

- TestContext: It is a class that organizes the test artifacts and contains test cases [11]. It can be stereotyped with “Variation Point”, which means that the test cases corresponding to the TestContext have variation points [11].
- TestCase: A test case is represented with UML sequence diagram in [11]. A test case can also be stereotyped as “Variation Point” for testing a functionality with variability [11].

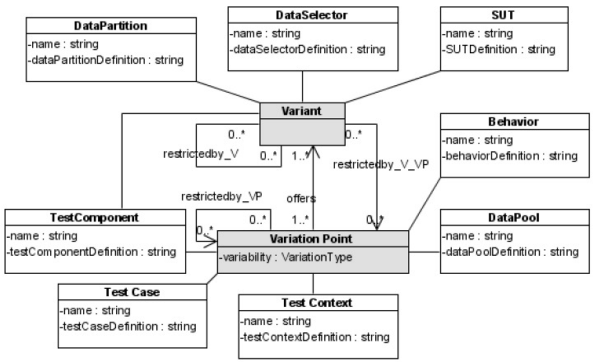


Figure 5. Proposed Extension to the UTP meta-model for handling variability [11]

- TestComponent: Test components interact with the SUT with the aim of realizing the test behaviour [11]. In the proposed extension, a test component can be stereotyped with “Variation Point” or “Variant”, which means that the test component can encapsulate the communication with the SUT, for the entire variation point or just for one of its variants.
- SUT: It can be stereotyped as “Variant”, which means that it realizes the functionality for its variant.
- DataPool, DataPartition and DataSelector: The DataPool contains the test data while the DataPartition the equivalence classes and data sets [11]. The dataPool can be stereotyped as “Variation Point”, which means that contains specific data for a Variation point. The DataPartition and the DataSelector are stereotyped as “Variant”, which means that the DataPartition contains the data associated with one of its variants and the dataSelector selects the data in the DataPartition for a specific variant.

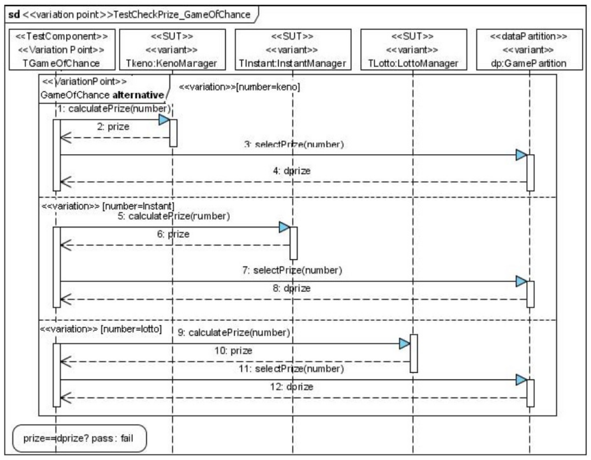

sd <<variation point>>TestCheckPrize_GameOfChance
<<TestComponent>> <<Variation Point>> TGameOfChance
<<SUT>> <<variant>> Tkeno:KenoManager
<<SUT>> <<variant>> TInstant:InstantManager
<<SUT>> <<variant>> TLotto:LottoManager
<<dataPartition>> <<variant>> dp:GamePartition
<<variationPoint>> GameOfChance alternative
<<variation>>[number=keno]
1: calculatePrize(number)
2: prize
3: selectPrize(number)
4: dprize
<<variation>> [number=Instant]
5: calculatePrize(rumber)
6: prize
7: selectPrize(number)
8: dprize
<<variation>> [number=lotto]
9: calculatePrize(number)
10: prize
11: selectPrize(number)
12: dprize
prize==dprize? pass : fail

Figure 6. Example of a Test Case Using the UTP Extention proposed in [11]

In [12], a model-based method for the automatic generation of test cases for the testing of SPLs is described using UML 2.0, the UML Testing Profile and the QVT language. The approach differentiates two main models: Platform Independent Models (PIM) and Platform Independent Test Models (PIT). Each of the models are separated for the domain engineering layer (PIMD and PITD) or the application engineering (PIM and PIT), as shown in Figure 7. Variability of the system models are managed with an extension of the UML Testing Profile. The proposed approach uses Orthogonal Variability Model (OVM) for managing variability of the test system. In this case, variability in the test system can be found in the test case behaviour and in the test architecture. The test case behaviour is modelled using sequence diagrams handling variability, whereas the variability-handling test architecture is modelled with UML class diagrams. The test model is automatically transformed taking as source models the design model and the variability model using QVT.

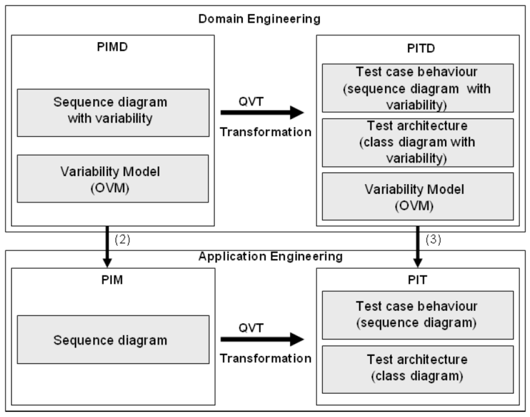

Domain Engineering
PIMD
Sequence diagram with variability
Variability Model (OVM)
QVT Transformation
PITD
Test case behaviour (sequence diagram with variability)
Test architecture (class diagram with variability)
Variability Model (OVM)
(2)
(3)
Application Engineering
PIM
Sequence diagram
QVT Transformation
PIT
Test case behaviour (sequence diagram)
Test architecture (class diagram)

Figure 7. Model driven testing approach for SPLs [12]

A Model-Based SPL regression testing approach is proposed in [13], where delta-oriented state machine as variable test models are used to incrementally evolve test artifacts by re-using artifacts of previously tested variants. Variability is applied in test artifacts, which are composed of (1) Test Models, (2) Test Goals, (3) Test Suite and (4) Test Plan. In [13], products evolve by applying deltas. Following the idea of Delta Modelling [18], a core model is developed and product variants are represented by the core model and a set of deltas that describe changes to be applied to this core model. From the testing point of view, a test artifact is developed to test the core system and deltas are applied to the test artifact in order to test the rest of the products.

Delta-oriented testing is also used in [14]. In this approach, Dukaczewski et al. propose a delta-oriented incremental testing approach based on textual requirements, where the test cases are directly associated to the requirements. Based on delta modelling [18], a delta describes how the behaviour of two system variants differs from each other. The main idea of this approach is based on testing the core system exhaustively and consider only the newly added or modified behaviour of the previous sytem during testing when moving to the next system variant [14]. Three steps are carried out to apply the proposed delta-oriented testing approach: The first step consist in selecting a core system, the requirements related to the selected system are separated from the requirements of all possible system variants and the selected core system is tested exhaustively. In the second a variant system is selected, deltas are applied to requirements to define changes between the requirements of different system variants by adding or removing requirements. Lastly, the test cases associated with the requirements are classified. Depending on the delta, the test cases are divided into four categories (Figure 8) [14]:

- Invalid: Test cases that become invalid for the new system variants. Invalid test cases belong to removed requirements.
- New: Test cases that are added to the new system variant, which belong to added requirements.
- Reuse: Test cases from the previous system variant that are not affected by the performed changes, which can be obtained using model slicing techniques (e.g. [19]). These test cases belong to unchanged requirements.
- Retest: Test cases from the previous system variant that are affected by the changes and have to be executed again. The test cases for retest are determined by the following identified options [14]: Random, Meta data, History/Statistics, Test expert and model.

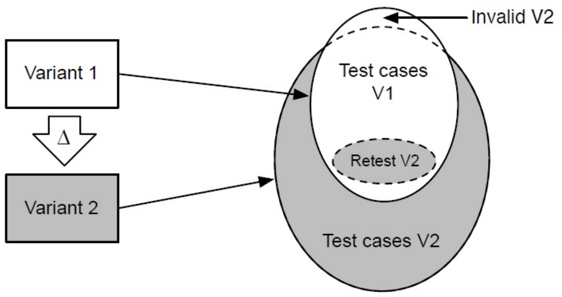


Figure 8. Delta Test-Sets [14]

B. Discussion and Analysis

The previous section has introduced the different documented approaches for handling variability in test systems. Each of these systems have their advantages and their limitations. When modelling variability in test systems, several characteristics are important, such as variability management, test automation, variability in test cases, or the specification of a test architecture. This section analyses the main limitations of each approach.

Kruse et al. propose a configurable test system for evolutionary testing of embedded systems in [8]. Variability in this test system appears in some configuration variables used by the evolutionary algorithm as well as the target configuration of the test system, i.e., MiL, SiL, PiL or HiL. Although the test system handles some variability points, it is not oriented for the validation of variability-handling systems.

Simulink is also used to model variability of test systems in two of the selected approaches ([9] [5]). This tool could be one of the most interesting when testing embedded software and CPSs, as it allows simulating the physical layer, as well as the cyber-digital layers (embedded system, software, etc.).

MiLEST is a toolbox oriented for the validation of embedded systems designed in [9], which can also be applied to CPSs. Although this test system shows variability modelling mechanisms, MiLEST is not designed for it. The variability points in the test architecture are limited to the test stimuli generator. Neither variability management tools nor automatic generation and configuration of the architecture for variability-intensive systems are used in this case. Another important factor when testing configurable systems is the simulation time. In this case, uniform variability is used as a variability modelling technique, which enlarges simulation time [16].

In our previous work [5], we analysed the variability of the test system and its components for the efficient validation of highly configurable CPSs. The test system is modelled in Simulink, and it is semi-automatically generated taking the information of a Feature Model into account. Apart from the SUT, variability can be found in the signals of the test data generator, requirements, test cases, signals of the test oracle, validation functions and validation function characteristics. In addition, we propose a traceability strategy among the features of the SUT and the test system. This strategy enables the automatic configuration of the test system depending on the selected SUT variant.

Apart from MATLAB/Simulink, other modelling languages are widely used when modelling embedded software, e.g., UML. In the case of [11], UML together with its UTP extension is used as a modelling tool. This approach analyses variability in several points, e.g., test context, test cases, etc. Moreover, some interesting concepts are provided that could be used in other test system, especially when modelling variability in test cases. In this case, variability is managed using UML models.

This UTP extension is used by the same author in [12]. In this case, the test models are generated automatically from the model of the SPL using the QVT Language. The variability modelling strategy of the test systems presented in [11] and [12] are clearly identified. In both cases, the test architecture is presented so that the interaction among the components of the test systems and the SUT is provided.

Delta modelling is used to model variability in [13] and [14]. In [13], Lity et al. propose a regression testing approach to test SPLs. The variability points of the test system can be found in test models, test goals, test suite and test plan. With regard to the drawbacks of this approach, on the one hand, the variability management of the test system is not specified. On the other hand, a test architecture is not considered, and as a consequence, the interaction among the components of the test system and the SUT cannot be appreciated.

Dukaczewski et al. propose incremental testing for the validation of SPLs in [14]. The way the test cases are classified in [14], “invalid”, “new”, “reuse” and “retest”, are an interesting option when testing SPLs. However, the proposed approach shows the same drawback as in [13], i.e., a variability management tool to model variability of the test system is not specified, and a test architecture showing the interactions among the test system and test components is not provided.

In the case of [10], a product line of validation environments is proposed. Although the interaction among the elements of the validation systems seems interesting, the analysed variability points of the systems are related to high-level elements, i.e., it does not take into account variability in test cases, test oracles, etc. Moreover, important characteristics such as variability management or test automation are not provided in this approach.

V. Open Challenges

Variability is an issue that has to be considered across the software-rich systems life cycle. In the previous section we have selected eight approaches that consider variability in the test systems. Nevertheless, there are still a lot of open questions and challenges when developing variant-rich test systems.

One of the major challenges would be to integrate a test system that could be able to test variant-rich systems from different domains. However, this is a complex issue, as different domains might need different kind of test systems. Furthermore, this can be infeasible because of the use of Domain Specific Languages, which warrants the use of different tools, e.g., SCADE for railway domain or MATLAB/Simulink for the automotive domain. Not only that, many variability points would have to be considered, as variability can appear in several points depending on the system.

Some of the selected test systems proposed a testing strategy. Depending on the validation phase, one test strategy could be more appropriate than another. Considering variability in the test strategy could be an interesting option, so that one test strategy or another could be chosen depending on the test needs and the validation stage.

A unified methodology that would warrant a systematic validation process of variant-rich systems of different domains could help test engineers with the validation activities. For instance, our previous work [20] proposes a model-based testing methodology for the validation of highly configurable CPSs.

One of the major problems in the validation of variant-rich systems is that as it is infeasible to test all the possible product configurations, the notion of the achieved test coverage is unclear. As a result, the analysis of new test metrics is a clear challenge in this field.

VI. Conclusion

This paper presents the current trends when modelling variability in test systems, issues that have to be considered as well as future challenges. Most of the research in the field of variability modelling of SPLs and variant-rich software focuses on the system itself. With regard to testing and validating SPLs and variant-rich software, most of the papers of the current state of the art propose generation of efficient configurations of the systems using different techniques such as combinatorial interaction testing (CIT). Other research efforts in this field consider the efficient test case generation for specific product variants. Although the research efforts in the field of variability modelling of test systems is not major, it is an important field of validation of variant-rich systems.

The paper has presented different approaches for testing different types of targets. In particular, the approaches presented in [9][8][5] are oriented to the testing and validation of real time embedded systems, embedded software or CPSs. On the other hand, the approaches presented in [11][12][13][14] have as testing objectives SPLs.

Some of the selected works do not consider variability management, e.g., [9][10]. This is not a problem if the test system is not variant rich, but in the case there are many variability points, the lack of a variability management tool can become a problem. In addition, the variability management tool can help to trace the variability of the test system with the variability of the SUT, as proposed in [5].

The paper also shows that variability in test systems is not just used to test variability-handling systems, but also general purpose systems. Three of the selected approaches consider variability in their test system although the SUT does not present any variability point. In the case of [8] and [9], the test systems are designed to test embedded systems, whereas the approach presented in [10] tests general purpose software.

VII. Acknowledgements

This work has been developed by the embedded systems group from Mondragon Goi Eskola Politeknikoa, supported by the Department of Education, Universities and Research of the Basque Government.

References

[1] 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.

[2] S. Thiel and A. Hein, "Modelling and using product line variability in automotive systems," IEEE Software, vol. 19, no. 4, 2002, pp. 66 – 72.

[3] 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.

[4] 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.

[5] A. Arrieta, G. Sagardui, and L. Etxeberria, "A configurable test architecture for the automatic validation of variability-intensive cyber-physical systems," in VALID 2014: The Sixth International Conference on Advances in System Testing and Validation Lifecycle, 2014, pp. 79–83.

[6] J. Zander-Nowicka, I. Schieferdecker, and P. J. Mosterman, A Taxonomy of Model-Based Testing for Embedded Systems from Multiple Industry Domains. Model-Based Testing for Embedded Systems, 2011, ch. 1, pp. 3–22.

[7] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering - a systematic literature review," Inf. Softw. Technol., vol. 51, no. 1, Jan. 2009, pp. 7–15.

[8] P. M. Kruse, J. Wegener, and S. Wappler, "A highly configurable test system for evolutionary black-box testing of embedded systems," in Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 1545–1552.

[9] J. Zander-Nowicka, "Model-based testing of real-time embedded systems in the automotive domain," Ph.D. dissertation, Technical University Berlin, 2008.

[10] 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.

[11] B. Pérez, M. Polo, and M. Piattini, "Towards an automated testing framework to manage variability using the uml testing profile," in AST, 2009, pp. 10–17.

[12] B. Pérez, M. Polo, and I. García, "Model-driven testing in software product lines," in Proceedings of the 2009 IEEE International Conference on Software Maintenance (ICSM 2009), 2009, pp. 511 – 514.

[13] S. Lity, M. Lochau, I. Schaefer, and U. Goltz, "Delta-oriented model-based spl regression testing," in 3rd International Workshop on Product LinE Approaches in Software Engineering, PLEASE 2012, Piscataway, NJ, USA, 2012, pp. 53 – 6.

[14] M. Dukaczewski, I. Schaefer, R. Lachmann, and M. Lochau, "Requirements-based delta-oriented spl testing," in 4th International Workshop on Product LinE Approaches in Software Engineering, PLEASE 2013, San Francisco, CA, United states, 2013, pp. 49 – 52.

[15] 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.

[16] A. Arrieta, G. Sagardui, and L. Etxeberria, "A comparative on variability modelling and management approaches in simulink for embedded systems," in V Jornadas de Computación Empotrada, ser. JCE 2014, no. 26-33, 2014.

[17] T. Thuem, C. Kastner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "Featureide: An extensible framework for feature-oriented software development," Science of Computer Programming, vol. 79, 2014, pp. 70 – 85.

[18] I. Schaefer, "Variability modelling for model-driven development of software product lines," in VaMoS, 2010, pp. 85–92.

[19] J. Kamischke, M. Lochau, and H. Baller, "Conditioned model slicing of feature-annotated state machines," in Proceedings of the 4th International Workshop on Feature-Oriented Software Development, ser. FOSD '12. New York, NY, USA: ACM, 2012, pp. 9–16.

[20] A. Arrieta, G. Sagardui, and L. Etxeberria, "A model-based testing methodology for the systematic validation of highly configurable cyber-physical systems," in VALID 2014: The Sixth International Conference on Advances in System Testing and Validation Lifecycle, 2014, pp. 66–72.

Automatic Test Set Generator with Numeric Constraints Abstraction for Embedded Reactive Systems: AUTSEG V2

Mariem Abdelmoula, Daniel Gaffé, and Michel Auguin

LEAT, University of Nice-Sophia Antipolis, CNRS
Email: Mariem.Abdelmoula@unice.fr
Email: Daniel.Gaffe@unice.fr
Email: Michel.Auguin@unice.fr

***Abstract*—AUTSEG is an automatic test set generator for embedded reactive systems. It automatically generates exhaustive test sets and allows to check safety properties of the tested system. A first version of AUTSEG has been initially designed for programs dealing with Boolean inputs and outputs. We present in this paper an extension of this tool called AUTSEG V2 to handle symbolic numeric data processing that provides more expressive and concrete tests of the system. To this end, we have developed a new library called superior linear decision diagrams (SupLDD) built on top of linear decision diagrams (LDD) library. This allows symbolic computation of system data while improving system verification (Determinism, Death sequences) and identifying all possible test cases. Our tool characterizes the system preconditions by numeric constraints to derive automatically the symbolic test cases using a backtracking operation. We demonstrate the application of AUTSEG V2 on an industrial example.**

Keywords–Test Sets; Synchronous Model; Pre-conditions; Numeric Data Processing; Backtrack; AUTSEG V2; SupLDD.

I. Introduction

Systems verification receives a particular interest today, especially for embedded reactive systems which have complex behaviors over time and which require long test sequences. This kind of systems is increasingly dominating safety critical domains such as nuclear industry, health insurance, banking, chemical industry, mining, avionics and online payment where failure could be disastrous. A practical solution in industry is to proceed using intensive test patterns in order to discover bugs, and increase confidence in the system, while researchers concentrate their efforts rather on formal verification. However, testing is obviously non exhaustive and formal verification is impracticable on real systems because of the combinatorial explosion nature of the states space.

AUTSEG [1] combines these two approaches to provide an automatic test set generator where formal verification ensures the automation in all phases of design, execution and test evaluation and help on get confidence in the consistency and relevance of tests. In a first version of AUTSEG, only Boolean inputs and outputs were supported while most of actual systems handle numeric data. Numeric data manipulation represents a big challenge for most of existing test generation tools due to the difficulty to express formal properties on those data using a concise representation. In our approach, we consider symbolic test sets which are thereby more expressive, safe and less complex than the concrete ones.

Therefore, we develop in this paper a new version of AUTSEG to take into account numeric data manipulation in addition to Boolean data manipulation. This was achieved by developing a new library for data manipulation called SupLDD. Prior automatic test sets generation methods have been consequently extended and adapted to this new numeric context. Symbolic data manipulations in AUTSEG V2 allow not only symbolic data calculations but also system verification (Determinism, Death sequences), and identification of all possible test cases without requiring the coverage of all the system states and transitions. Therefore, our approach bypasses in numerous cases the states space explosion problem. We besides defined a backtrack operation to exhibit significant test sets of the target system.

In the remainder of this paper, we briefly recall the principles of AUTSEG V1 and introduce the new version AUTSEG V2. The principles of data manipulation and its capabilities on tests generation and verification are presented in Section II. A case study is presented in Section III. We show in Section IV experimental results. Finally, we conclude the paper in Section V with some directions for future works.

II. AUTSEG V2 Description

A. Architectural Test Overview

We introduce in this section the principles of our automatic testing approach including data manipulation. Figure 1 shows five main operations including: i) the design of a global model of the system under test, ii) a quasi-flattening operation, iii) a compilation process, iv) a generation process of symbolic sequences mainly related to the symbolic data manipulation entity, v) and finally the backtrack operation to generate all possible test cases.

In this paper, we particularly focus on verification of embedded software controlling reactive systems behavior. The conception of such systems is generally based on the synchronous approach [2] that presents clear semantics to exceptions, delays and actions suspension. This notably reduces the programming complexity and favors the application of verification methods. In this context, we present the global model by hierarchical and parallel concurrent Finite States Machines (FSMs) based on the synchronous approach. The hierarchical machine describes the global system behavior, while parallel automata act as observers for control data of

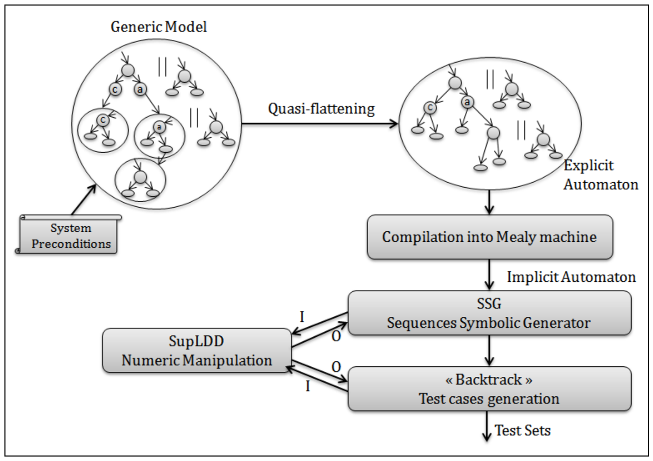


Figure 1. Global test process.

the hierarchical automaton. Our approach allows to test many types of a system at once. In fact, we present a single generic model for all types of the system, the specification of tests can be done later using particular Boolean variables called system preconditions (type of system, system mode, etc.). Hence, a specific test generation could be done at the end of test process through analysis of the system preconditions. This prevents to generate as many models as system types, which can highly limit the legibility and increase the risk of specification bugs.

A straightforward way to analyze a hierarchical machine is to flatten it first (by recursively substituting in a hierarchical FSM each super state with its associated FSM) and then apply on the resulting FSM a verification tool such as a model cheking tool. However, to analyze the global model, a full flattening of the hierarchical FSM is not required. Only the sequential hierarchical automata is flattened, the global structure remains parallel. In fact, flattening parallel FSMs explodes usually in number of states. Thus there is no need to flatten them, as we can compile them separately thanks to the synchronous approach [2], then concatenate them with the flat model retrieved at the end of the compilation process. This quasi-flattening operation allows to flatten the hierarchical automata and maintain the parallelism. This offers a simpler model, a faster compilation, and brings more flexibility to identify all possible evolutions of the system as detailed in the following steps.

Resulting flat automata and concurrent automata are then compiled separately into explicit Mealy machines, implicitly represented by a set of Boolean equations. Compilation results of these automata are concatenated at the end of this process. They are represented by a union of sorted equations rather than a Cartesian product of graphs to support the synchronous parallel operation and instantaneous diffusion of signals as required by the synchronous approach. Accordingly, a substantial reduction is brought on the size of the system model. Our compilation requires only $log_2(nbstates)$ registers, while classical works uses one register per state [3]. It allows also checking the determinism of all automata which ensures the persistence of the system behavior.

To supply numeric data manipulation in our tests, we developed SupLDD library offering symbolic means to characterize several preconditions by numeric constraints. It is sorely based on the potency of LDD library [4]. The symbolic representation of these preconditions shows an important role in the following operations of sequences symbolic generation and test cases generation ”Backtrack”. It evenly enhances system security by analyzing the constraints computations.

During the sequences symbolic generation operation, we automatically extract necessary preconditions which lead to specific, significant states of the system from generated sequences. Having defined the optimal preconditions for restricting the states space, we work locally on significant subspaces. This sequences generation process relies on the effective representation of the global model and the robustness of numeric data processing to generate the exhaustive list of possible sequences, avoiding therefore the manual and explicit presentation of all possible combinations of system commands.

Finally, the verification of the whole system behavior is performed by the manipulation of extracted preconditions from each significant subspace. Namely, we verify the execution context of each significant subspace. This verification is performed by the backtrack operation. It generates all possible test cases of the system under test. Specifically, it identifies all paths satisfying each final critical state preconditions to reach the root state.

We have already detailed in [1] the principles of the global model conception, the quasi-flattening operation and the compilation process. We will rather focus in the rest of this paper on the presentation of symbolic data manipulations and their capabilities to carry the symbolic sequences generation and the backtrack operation.

B. *Symbolic data manipulation*

1) Related work: Since 1986, Binary Decision Diagrams (BDDs) have successfully emerged to represent Boolean functions for formal verification of systems with large states space. BDDs, however, cannot represent quantitative information such as integers and real numbers. Variations of BDDs have been proposed thereafter to support symbolic data manipulations that are required for verification and performance analysis of systems with numeric variables. For example, Multi-Terminal Binary Decision Diagrams (MTBDDs) [5] are a generalization of BDDs in which there can be multiple terminal nodes, each labeled by an arbitrary value. However, the size of nodes in an MTBDD can be exponential (2^n) for systems with large range of values. To support a larger number of values, Yung-Te Lai has developed Edge-Valued Binary Decision Diagrams (EVBDDs) [6] as an alternative to MTBDDs to offer a more compact form. EVBDDs associate multiplicative weights with the true edges of an EVBDD function graph to allow an optimal sharing of subgraphs. This suggests a linear evolution of non-terminal nodes size rather than an exponential one for MTBDDs. However, EVBDDs are limited to relatively simple calculations units, such as adders and comparators, implying a high cost per node for complex calculations such as $(X \times Y)$ or (2^X).

To overcome this exponential growth, Binary Moment Diagrams (BMDs) [7], another variation of BDDs, have been specifically developed for arithmetic functions considered as linear functions with Boolean inputs and integer outputs to perform a compact representation for integer encodings and operations. They integrate a moment decomposition principle giving way to two sub-functions representing the two moments (constant and linear) of the function, instead of a decision. This representation was later extended to Multiplicative Binary Moment Diagrams (*BMDs) [8] to include weights on edges allowing to share common sub-expressions. These edges weights are multiplicatively combined in a *BMD, in contrast to the principle of addition in an EVBDD. Thus, the following arithmetic functions $X + Y$, $X - Y$, $X \times Y$, 2^X show representations of linear size. Despite their significant success in several cases, handling edges weights in BMDs and *BMDs is a costly task. Moreover, BMDs are unable to verify the satisfiability property, and functions outputs are non divisible integers to separate bits, causing a problem for applications with output bit analysis. BMDs and MTBDDs were combined by Clarke and Zhao in Hybrid Decision Diagrams (HDDs) [9]. But, all of these diagrams are restricted to materials arithmetic circuits check and not suitable for the verification of software systems specifications.

Within the same context of arithmetic circuits check, Taylor Expansion Diagrams (TEDs) [10] have been introduced to supply a new formalism for multi-values polynomial functions providing a more abstract, standard and compact design representation, with integer or discrete inputs and outputs values. For an optimal fixed order of variables, the resulting graph is canonical and reduced. Unlike the above data structures, TED is defined on a non-binary tree. In other words, the number of child nodes depends on the degree of the relevant variable. This makes TED a complex data structure for particular functions such as (a^x). In addition, the representation of the function $(x < y)$ is an important issue in TED. This is particularly challenging for the verification of most software systems specifications. In this context, Decision Diagrams for Difference logic (DDDs) [11] have been proposed to present functions of first order logic by inequalities of the form $\{x - y \leq c\}$ or $\{x - y < c\}$ with integer or real variables. The key idea is to present these logical formulas as BDD nodes labeled with atomic predicates. For a fixed variables order, a DDD representing a formula f is no larger than a BDD of a propositional abstraction of f. It supports as well dynamic programming by integrating an algorithm called QELIM based on Fourier-Motzkin elimination [12]. Despite their proved efficiency in verifying timed systems [13], the difference logic in DDDs is too restrictive in many program analysis tasks. Even more, dynamic variable ordering (DVO) is not supported in DDDs. To address those limitations, LDDs [4] extend DDDs to full Linear Arithmetic by supporting an efficient scheduling algorithm and a QELIM quantification. They are BDDs with non terminal nodes labeled by linear atomic predicates satisfying a scheduling theory and local constraints reduction. Data structures in LDDs are optimally ordered and reduced by considering the several implications of all atomic predicates. LDDs have the possibility of computing arguments that are not fully reduced or canonical for most LDD operations. This suggests the use of various reduction heuristics that trade off reduction potency for calculations cost.

2) SupLDD: We summarize from the above data structures that LDD is the most relevant work for data manipulation in our context. We present in this section a new library for data manipulation called SupLDD founded on LDD basis. Figure 2 shows an example of representation in SupLDD of the arithmetic formula $F1 = \{(x \geq 5) \wedge (y \geq 10) \wedge (x + y \geq 25)\} \vee \{(x < 5) \wedge (z > 3)\}$. Nodes of this structure are labeled by the linear predicates $\{(x < 5); (y < 10); (x + y < 25); (-z < -3)\}$ of formula F1, where the right branch evaluates its predicates to 1 and the left branch evaluates its predicates to 0. In fact, the choice of a particular comparison operator within the 4 possible operators $\{<, \leq, >, \geq\}$ is not important since the 3 other operators can always be expressed from the chosen operator: $\{x < y\} \Leftrightarrow \{NEG(x \geq y)\}; \{x < y\} \Leftrightarrow \{-x > -y\}$ and $\{x < y\} \Leftrightarrow \{NEG(-x \leq -y)\}$.

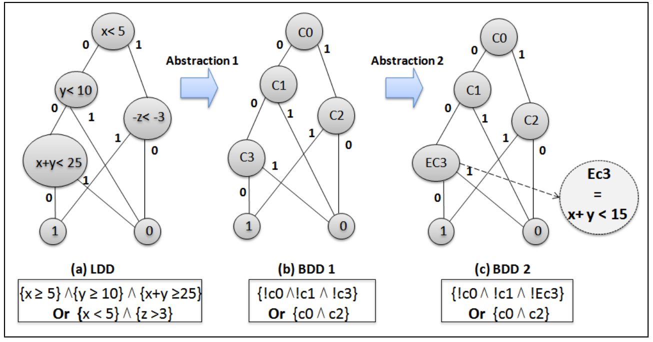

x< 5
y< 10
-z< -3
x+y< 25
0
1
Abstraction 1
C0
C1
C2
C3
Abstraction 2
EC3
Ec3 = x+ y < 15
(a) LDD
{x ≥ 5} ∧{y ≥ 10} ∧ {x+y ≥25} Or {x < 5} ∧ {z >3}
(b) BDD 1
{!c0 ∧!c1 ∧ !c3} Or {c0 ∧ c2}
(c) BDD 2
{!c0 ∧ !c1 ∧ !Ec3} Or {c0 ∧ c2}

Figure 2. Representation in SupLDD of F1.

We show in Figure 2.b that the representation of F1 in SupLDD has the same structure as a representation in BDD that labels its nodes by the corresponding Boolean variables $\{C0; C1; C2; C3\}$ to each SupLDD predicate. But, a representation in SupLDD is more advantageous. In particular, it ensures the numeric data evaluation and manipulation of all predicates along the decision diagram. This furnishes a more accurate and expressive representation in Figure 2.c than the original BDD representation. Namely, the Boolean variable C3 is replaced by EC3 which evaluates the corresponding node to $\{x+y < 15\}$ instead of $\{x+y < 25\}$ taking into account prior predicates $\{x < 5\}$ and $\{y < 10\}$. Besides, SupLDD relies on an efficient T-atomic scheduling algorithm [4] that makes compact and non-redundant diagrams for SupLDD where a node labeled for example by $\{x \leq 15\}$ never appears as a right child of a node labeled by $\{x \leq 10\}$. As well, nodes are ordered by set of atoms $\{x, y, etc.\}$ where a node labeled by $\{y < 2\}$ never appears between two nodes labeled by $\{x < 0\}$ and $\{x < 13\}$. Further, SupLDD diagrams are optimally reduced including the LDD reduction rules. First, the QELIM quantification introduced in LDDs allows the elimination of multiples variables: For example, the QELIM quantification of the expression $\{(x-y \leq 3) \wedge (x-t \geq 8) \wedge (y-z \leq 6) \wedge (t-k \geq 2)\}$ eliminates the intermediate variables *y* and *t* and generates the simplified expression $\{(x - z \leq 9) \wedge (x - k \geq 10)\}$. Second, the LDD high implication [4] rule allows to get the smallest geometric space: For example the simplification of the expression $\{(x \leq 3) \wedge (x \leq 8)\}$ in high implication turns to the single term $\{x \leq 3\}$. Finally, the LDD low implication [4] rule generates the largest geometric space where the expression $\{(x \leq 3) \wedge (x \leq 8)\}$ becomes $\{x \leq 8\}$.

SupLDD operations- SupLDD operations are primarily generated from basic LDD operations [4]. They are simpler and more adapted to our needs. We present functions to manipulate inequalities of the form $\{\sum a_i x_i \leq c\}$; $\{\sum a_i x_i < c\}$; $\{\sum a_i x_i \geq c\}$; $\{\sum a_i x_i > c\}$; where $\{a_i, x_i, c \in Z\}$. Given two inequalities I_1 and I_2, the main operations in SupLDD include:

- SupLDD conjunction (*I1*, *I2*): This absolutely corresponds to the intersection on Z of sub-spaces representing *I1* and *I2*.
- SupLDD disjunction (*I1*, *I2*): As well, this operation absolutely corresponds to the union on Z of sub-spaces representing *I1* and *I2*.

Accordingly, all the space Z can be represented by a union of two inequalities $\{x \leq a\} \cup \{x > a\}$. As well, the empty set can be inferred from the intersection of inequalities $\{x \leq a\} \cap \{x > a\}$.

- Equality operator $\{\sum a_i x_i = c\}$: It is defined by the intersection of two inequalities $\{\sum a_i x_i \leq c\}$ and $\{\sum a_i x_i \geq c\}$.
- Resolution operator: It simplifies arithmetic expressions using QELIM quantification, and both low and high implication rules introduced in LDD. For example, the QELIM resolution of $\{(x-y \leq 3) \wedge (x-t \geq 8) \wedge (y-z \leq 6) \wedge (x-t \geq 2)\}$ gives the simplified expression $\{(x-z \leq 9) \wedge (x-t \geq 8) \wedge (x-t \geq 2)\}$. This expression can be more simplified to $\{(x-z \leq 9) \wedge (x-t \geq 8)\}$ in case of high implication and to $\{(x-z \leq 9) \wedge (x-t \geq 2)\}$ in case of low implication.
- Reduction operator: It solves an expression A with respect to an expression B. In other words, if A implies B, then the reduction of A with respect to B is the projection of A when B is true. For example, the projection of A $\{(x-y \leq 5) \wedge (z \geq 2) \wedge (z-t \leq 2)\}$ with respect to B $\{x-y \leq 7\}$ gives the reduced set $\{(z \geq 2) \wedge (z-t \leq 2)\}$.

We report in this paper on the performance of these functions to enhance our tests. More specifically, by means of SupLDD library, we present next an extension of Sequences Symbolic Generation operation initially presented in AUTSEG V1 to integrate data manipulation and generate more significant and expressive sequences. Moreover, we track and analyze tests execution to spot the situations where the program violates its properties. In the other hand, our library ensures the analyze of generated sequences context to carry the backtrack operation and generate all possible test cases.

C. Sequences Symbolic Generation (SSG)

In this version, we take into account data calculations within the sequences generation process. Let's recall the principles of SSG in AUTSEG V1. In fact, our approach is primarily designed to test systems running iterative commands. In this context, we confine only on significant sub-spaces representing each command of the system instead of considering all the states space. Indeed, we test all the system commands, but one command is tested at once. This restriction was done by characterizing all preconditions defining the execution context in each subspace. Hence, the major complex calculation is intended to be locally done in each significant subspace avoiding the states space combinatorial explosion problem. Figure 3 shows this efficient representation of the system behavior. It presents a repetition of a subspace pattern representing a specific system command instead of an infinite tree if we typically imagine all possible combinations of the system iterative commands.

Each state in the subspace is specified by 3 main variables: symbolic values of the program variables, path condition

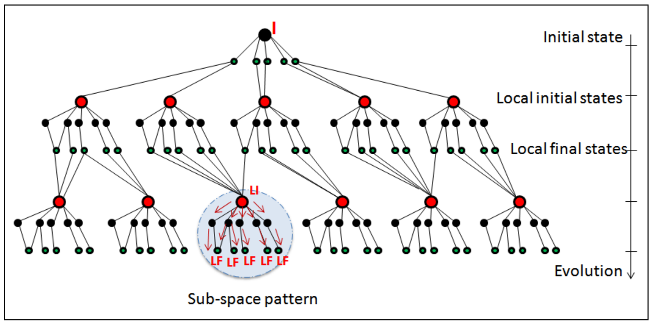


Figure 3. AUTSEG V2 Model Representation.

and command parameters (next byte-code to be executed). The path condition represents preconditions that should be satisfied by the symbolic values to successfully progress the execution of the current path. We particularly define two types of preconditions:

- Boolean global preconditions that define the execution context of a given command. They states the list of commands that should be executed before. They arise as command output if this latter is properly executed.
- Numeric local preconditions that define numeric constraints on commands parameters. They are presented and manipulated by SupLDD functions.

We have explained in details in the first version of AUTSEG the SSG operation. We have applied BDD-analysis to generate all possible paths from a Local Initial state (*LI*) to reach Local Final states (*LF*) of the tested subspace. As well, necessary preconditions are extracted from this subspace check. We extend in this paper the SSG operation to integrate data manipulations. We apply SupLDD analysis on numeric local preconditions to check if the tested system is safe. We firstly check if there are erroneous sequences. To this end, we apply the SupLDD conjunction function on all extracted numeric preconditions within the analyzed path. If the result of this conjunction is null, the analyzed sequence is then impossible and should be rectified! Second, we check the determinism of the system behavior. To this end, we verify if the SupLDD conjunction of all outgoing transitions from each state is empty. In other words, we verify if the SupLDD disjunction of all outgoing transitions from each state is equal to all the space covering all possible system behaviors.

Contrary to the classical sequences generator, our tool constantly generates a tree of pure future states, thus preventing loops from occurring. Namely, previous states always converge to the global initial state. This approach easily favors the backtrack execution.

D. Backtrack operation

Once the necessary preconditions are extracted, a next step is to backtrack paths from each final critical state until the initial state finding the sequence fulfilling these preconditions. This operation is carried by robust calculations on SupLDD and the compilation process which kept enough knowledge to find later the previous states. It includes two main actions: a global backtrack and a local backtrack. Let's consider the SSG representation in Figure 3, if we take into account *LF* as a critical final state *FS* of the tested system, the global backtrack operation is to find the list of commands that should be executed before the tested command. Figure 4 details this operation: Given the global extracted preconditions (*GP1*,*GP2*, etc.) from the SSG operation at this level (Final state *FS* of command *C1*), we search in the global actions table for actions (Commands *C2* and *C3*) that emit each parsed global precondition. Next, we put on a list *SL* the states that trigger each identified action (*SL*= $\{C2, C3\}$). This operation is iteratively executed on all found states (*C2,C3*) until reaching the root state *I* with zero preconditions (*C4* with zero preconditions).

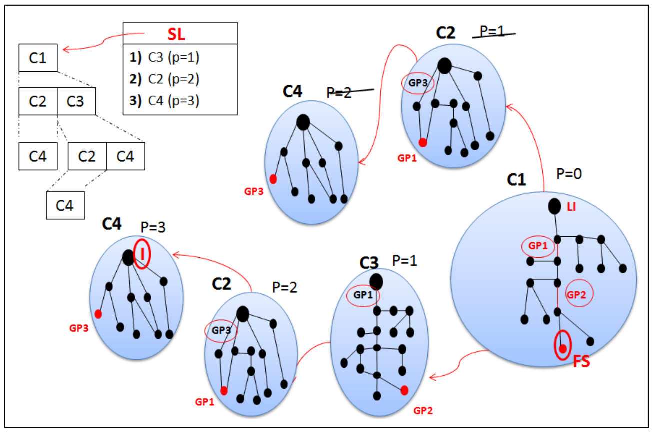


Figure 4. Global Backtrack.

As many commands can share the same global preconditions (*C1* and *C3* share the same precondition *GP1*), the identified states can be repeated on *SL* (*C2* and *C4* are repeated on *SL*). To manage this redundancy, we allocate a priority *P* to each found state where each state of priority *P* should precede the state of priority *P+1*. More specifically, if an identified state already exists in *SL*, then its priority is incremented by 1 (Priority of *C2* and *C4* are incremented by 1). By the end of this operation, we obtain the list *SL* (*SL*= { *C3,C2,C4* }) of final states refering to subspaces that should be traced to reach *I*.

A next step is to execute a local backtrack on each identified subspace (*C1,C3, C2,C4*) starting from the state with the lowest priority and so on to trace the final path from *FS* to *I*. The sequence from *I* to *FS* is an example of a good test set. Figure 5 presents an example of local backtrack in command *C3*. In fact, during the SSG operation each state *S* was labeled by (1) a Local numeric Precondition (*LP*) presenting numeric constraints that should be satisfied on its ongoing transition and (2) a Total Local numeric precondition (*TL*) that presents the conjunction of all *LP* along the executed path from *I* to *S*. To execute the local backtrack, we start from the ongoing transition *PT* to *FS* to find a path that satisfy the backtrack precondition *BP* initially defined by *TL*. If the backtrack precondition is satisfied by the total precondition $\{TL \geq BP\}$, then if the local precondition *LP* of the tested transition is not null, So we remove this verified precondition *LP* from *BP* by applying the SupLDD projection function. Next, we move to the amount state of *PT* and test its ongoing transitions, etc. However, if $\{TL < BP\}$, we move to the test of other ongoing transitions to find the transition from which *BP* can be satisfied. This operation is iteratively executed until reaching the initial state on which the backtrack precondition

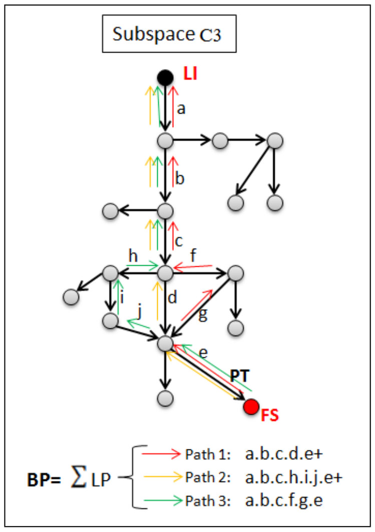


Figure 5. Local Backtrack.

is null (fully satisfied). In short, if the context is verified, the generated sequence is considered correct. At the end of this process, we join all identified paths from each traced subspace according to the given priority order from the global backtrack operation.

III. USE CASE

To illustrate our approach, we studied the case of a contactless smart card for the transportation sector manufactured by ASK company [14], a world leader in contactless smart card technology. We specifically target the verification of the card's functionality and security features. Overall, security of such systems is critical: it can concern cards for access security, banking, ID, etc. Cards complexity makes it difficult for a human to identify all possible sensitive situations or to validate it by classical methods. We need approximately 500 000 years to test the first 8 bytes if we consider a classical Intel processor able to generate 1000 test sets per second. As well, combinatorial explosion of possible modes of operation makes it nearly impossible to attempt a comprehensive simulation. The problem is exacerbated when the system integrates numeric data processing. We have already studied this use case within the first version of AUTSEG, but processing numeric variables was ignored. We rather show in this section real tests with AUTSEG V2 taking into account the complexity of data manipulation.

The smart card operation is defined by a transport standard called Calypso that presents 33 commands. The succession of these commands (e.g., Open Session, SV Debit, Get Data, Change Pin) gives the possible scenarios of card operation. We designed the generic model of the studied card by 52 interconnected automata including 765 states. Forty three of them form a hierarchical structure. The remaining automata operate in parallel and act as observers to control the global context of hierarchical automaton (Closed Session, Verified PIN, etc.). We choose to use Light Esterel (light version of SyncChart) [15], a synchronous graphical model that integrates high-level concepts of synchronous languages in an expressive graphical formalism. We show in Figure 6 a small part of our model representing the command Open Session. Each command in Calypso is presented by an APDU (Application Protocol Data Unit) that presents the next byte-code to be executed (CLA,INS,P1,P2, etc.). We expressed these parameters by SupLDD local preconditions on various transitions. For instance, AUTSEGINT($h10 < P1 < h1E$) means that the corresponding transition can only be executed if ($10 < P1 < 30$). Back-Autseg-Open-Session and Back-Autseg-Verify-PIN are examples of global preconditions that appear as outputs of respectively Open Session and Verify PIN commands when they are correctly executed. They appear also as inputs for other commands as SV Debit command to denote that the card can be debited only if the PIN code is correct and a session is already open. Autseg-Contact-mode is an example of system precondition specifying that Open Session command should be executed in a Contactless Mode.

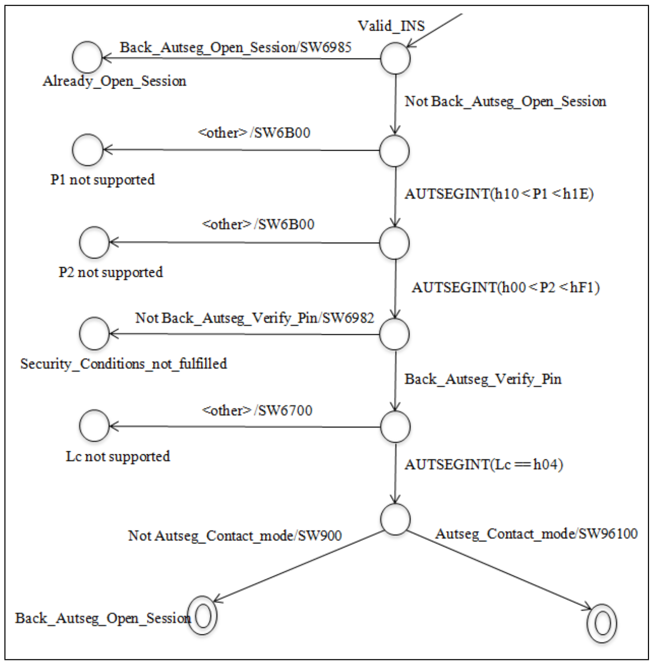

Valid_INS
Back_Autseg_Open_Session/SW6985
Already_Open_Session
Not Back_Autseg_Open_Session
<other> /SW6B00
P1 not supported
AUTSEGINT(h10 < P1 < h1E)
<other> /SW6B00
P2 not supported
AUTSEGINT(h00 < P2 < hF1)
Not Back_Autseg_Verify_Pin/SW6982
Security_Conditions_not_fulfilled
Back_Autseg_Verify_Pin
<other> /SW6700
Lc not supported
AUTSEGINT(Lc == h04)
Not Autseg_Contact_mode/SW900
Autseg_Contact_mode/SW96100
Back_Autseg_Open_Session

Figure 6. Open Session command.

IV. Experimental Results

In this section, we show experimental results of applying our tool to the contactless transportation card. We intend to test the security of all possible combinations of 33 commands of the Calypso standard. This validation process is extremely important to determine whether the card correctly meets its specifications. Each command in the Calypso standard is encoded on a minimum of 8 bytes. We conducted our experiments on a PC with Intel Dual Core Processor, 2 GHz and 8 GB RAM. We have already shown in a previous work [1] the successful application of our quasi-flattening process on the smart card hierarchical model. Compared to classical works, we have moved from 9.6 10^{24} states in the designed model to only 256 per branch of parallel. Then, due to the compilation process, we have moved from 477 registers to only 22.

We present in this paper more interesting results on sequences generation and test coverage with data processing. The curve denoted C1 in Figure 7 shows an exponential evolution of the number of generated sequences versus the number of tested bytes. This corresponds to a classical testing method that browses all possible paths of the card model without any restriction. We are not even able to test more than 2 commands of the model. Our model explodes by 13 bytes generating 3,993,854,132 possible sequences. A second test applies AUTSEG V1 on the card model represented in the same manner as Figure 3. Results shows in curve C2 a lower evolution that stabilizes at 10 steps and 1784 paths, allowing for coverage of all states of the tested model. More interesting results are shown in curve C3 by AUTSEG V2 tests. Our approach enables coverage of the global model in a substantially short time (few seconds). It allows separately testing 33 commands (all the system commands) in only 21 steps, generating a total of solely 474 paths. Covering all states in only 21 steps, our results demonstrate that we test separately one command (8 bytes) at once in our approach thanks to the backtrack operation. The additional steps (13 bytes) correspond to the test of system preconditions (e.g., Autseg-Contact-mode, etc.), global preconditions (Back-Autseg-Open-Session, etc.) and other local preconditions (e.g., AUTSEGINT($h00 \leq buffer-size \leq hFF$)). Whereas, only fewer additional steps (2 bytes) are required within the first version of AUTSEG that stabilizes at 10 steps. This difference proves a complete evaluation of system constraints by our new version of AUTSEG performing therefore more expressive and reel tests: we integrate a better knowledge of the system.

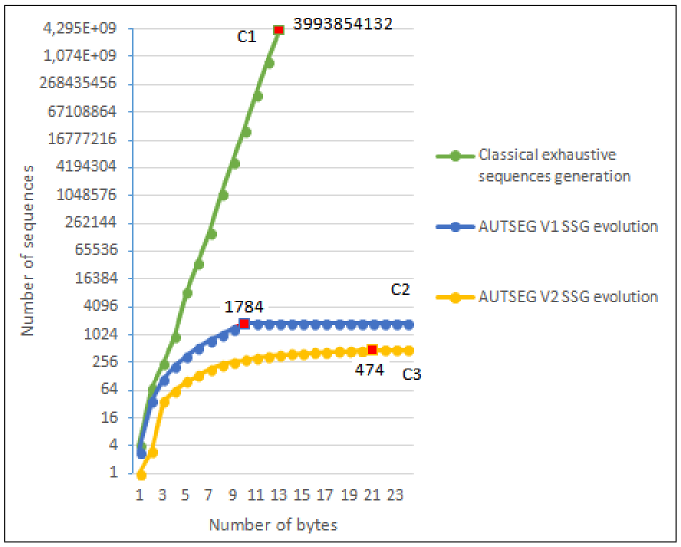


Figure 7. SSG evolutions.

Curve C4 in Figure 8 exhibits results of AUTSEG V2 tests simulated with 3 anomalies on the smart card model. We note less generated sequences by the 5 steps. We obtain a total of 460 sequences instead of 474 at the end of tests. 14 sequences are removed since they are unfeasible (dead sequences) by SupLDD calculations. Indeed, the SupLDD conjunction of parsed local preconditions AUTSEGINT($01h \leq RecordNumber \leq 31h$) and AUTSEGINT($RecordNumber \geq FFh$) within a same path is null presenting an over-specification example (anomaly) of the Calypso standard that should be revised.

We show in Figure 9 an excerpt of generated sequences

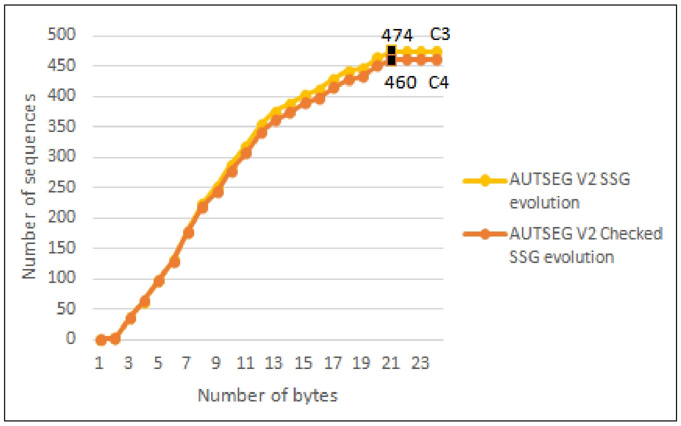


Figure 8. AUTSEG V2 SSG evolutions.

by AUTSEG V2 detecting another type of anomaly: an under-specification in the card behavior. The *Incomplete Behavior* message reports a missing action on a tested state of Update-Binary command. Indeed, two actions are defined ($Tag = 54h$) and ($Tag = 03h$) at this state. All states where Tag is different from *84* and *3* are missing. We can automatically spot such problems by checking for each parsed state if the union SupLDD-Or of all outgoing transitions is equal to all the space. Once, this property is always true, then the smart card behavior is proved deterministic.

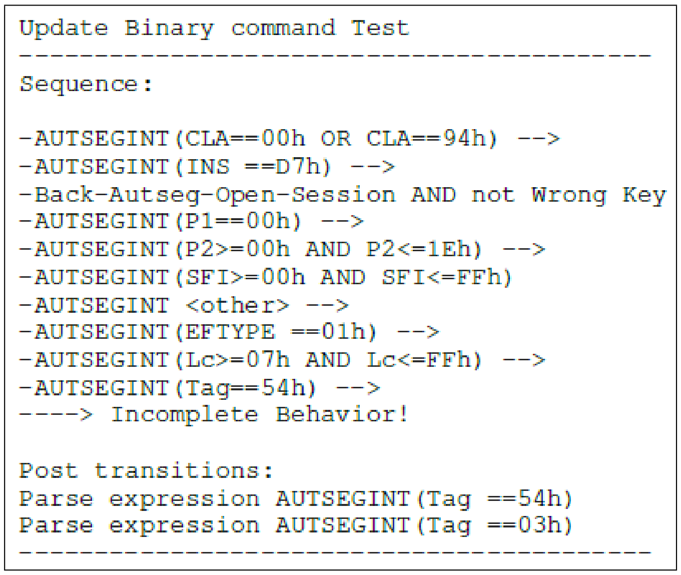

```
Update Binary command Test
------------------------------------------
Sequence:

-AUTSEGINT(CLA==00h OR CLA==94h) -->
-AUTSEGINT(INS ==D7h) -->
-Back-Autseg-Open-Session AND not Wrong Key
-AUTSEGINT(P1==00h) -->
-AUTSEGINT(P2>=00h AND P2<=1Eh) -->
-AUTSEGINT(SFI>=00h AND SFI<=FFh)
-AUTSEGINT <other> -->
-AUTSEGINT(EFTYPE ==01h) -->
-AUTSEGINT(Lc>=07h AND Lc<=FFh) -->
-AUTSEGINT(Tag==54h) -->
----> Incomplete Behavior!

Post transitions:
Parse expression AUTSEGINT(Tag ==54h)
Parse expression AUTSEGINT(Tag ==03h)
------------------------------------------
```

Figure 9. Smart Card Under-specification.

As explained before, we get the execution context of each generated sequence at the end of this operation. The next step is then to backtrack all critical states of the Calypso standard (all final states of 33 commands). We show in Figure 10 a detailed example of backtrack from the final state of SV Undebit command that emit SW6200 code.

We identify from the global extracted preconditions Back-Autseg-Open-Session and Back-Autseg-Get-SV the list of commands (Open Secure Session and SV Get) to be executed before. Then, we look recursively for all global preconditions

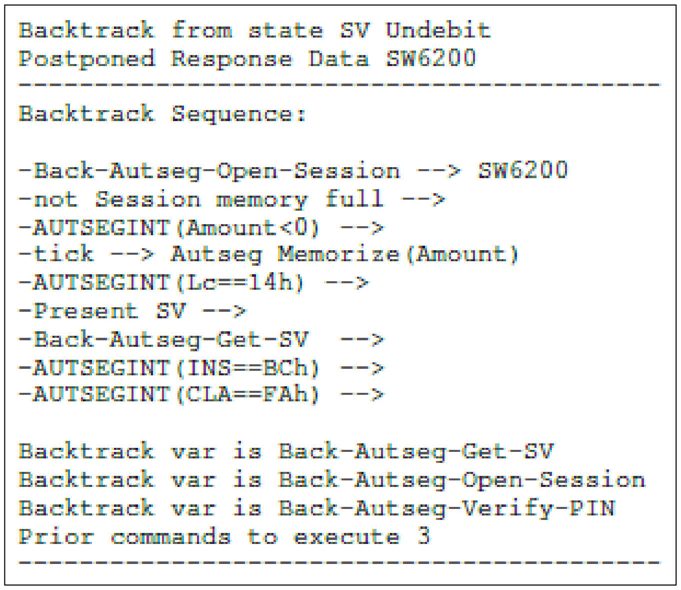

```
Backtrack from state SV Undebit
Postponed Response Data SW6200
------------------------------------------
Backtrack Sequence:

-Back-Autseg-Open-Session --> SW6200
-not Session memory full -->
-AUTSEGINT(Amount<0) -->
-tick --> Autseg Memorize(Amount)
-AUTSEGINT(Lc==14h) -->
-Present SV -->
-Back-Autseg-Get-SV -->
-AUTSEGINT(INS==BCh) -->
-AUTSEGINT(CLA==FAh) -->

Backtrack var is Back-Autseg-Get-SV
Backtrack var is Back-Autseg-Open-Session
Backtrack var is Back-Autseg-Verify-PIN
Prior commands to execute 3
------------------------------------------
```

Figure 10. SV Undebit Backtrack.

of each identified command to trace the complete path to the initial state of Start command. We observe from the results that Verify PIN command should proceed the Open Secure Session command. So, the final backtrack path is to trace (local backtrack) the identified commands respectively SV Undebit, SV Get, Open Secure Session et Verify PIN using local preconditions of each command.

At the end of this process, we generate automatically 5456 test sets that cover the entire behavior of the studied smart card. While, industrials take much more time to solely generate manually 520 test sets covering 9,5% of our tests as shown in Figure 11.

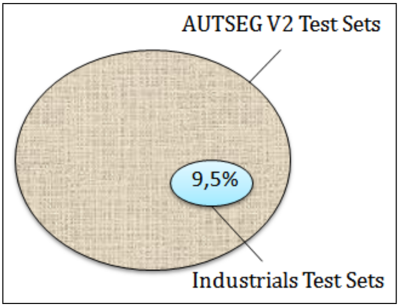


Figure 11. Tests Coverage.

V. CONCLUSION

We have proposed an extension of AUTSEG to integrate data manipulations. For this purpose, we have developed a new library called SupLDD that supplies numeric data manipulations and takes advantages of the symbolic encoding scheme of BDDs. This library allows not only symbolic calculations of system data but also the verification of the system behavior. Our method is practical and performs well, even with large models where the risk of combinatorial explosion of states space is important. Our experiments confirm that our tool provides more expressive and significant tests covering all possible system evolutions in a short time. More generally, our tool including the SupLDD calculations can be applied to many numeric systems as they could be modeled by FSMs handling integer variables.

Since SupLDD is implemented on top of a simple BDD package. We aim in a future work to rebuild SupLDD on top of an efficient implementation of BDDs with complement edges [16] to get a better optimization of our library. Another interesting contribution would be to integrate SupLDD in data abstraction of CLEM [15]. More details about these future works are presented in [17].

References

[1] M. Abdelmoula, D. Gaffé, and M. Auguin, "Autseg: Automatic test set generator for embedded reactive systems," in Testing Software and Systems, 26th IFIP International Conference,ICTSS, ser. Lecture Notes in Computer Science. Madrid, Spain: springer, September 2014, pp. 97–112.

[2] C. André, "A synchronous approach to reactive system design," in 12th EAEEIE Annual Conf., Nancy (F), May 2001, pp. 349–353.

[3] I. Chiuchisan, A. D. Potorac, and A. Garaur, "Finite state machine design and vhdl coding techniques," in 10th International Conference on development and application systems. Suceava, Romania: Faculty of Electrical Engineering and Computer Science, 2010, pp. 273–278.

[4] S. Chaki, A. Gurfinkel, and O. Strichman, "Decision diagrams for linear arithmetic." in FMCAD. IEEE, 2009, pp. 53–60.

[5] M. Fujita, P. C. McGeer, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation," Form. Methods Syst. Des., vol. 10, no. 2-3, Apr. 1997, pp. 149–169.

[6] Y.-T. Lai and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," in Proceedings of the 29th ACM/IEEE Design Automation Conference, ser. DAC'92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 608–613.

[7] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference, ser. DAC '95. New York, NY, USA: ACM, 1995, pp. 535–541.

[8] L. Arditi, "A bit-vector algebra for binary moment diagrams," I3S, Sophia-Antipolis, France, Tech. Rep. RR 95–68, 1995.

[9] E. Clarke and X. Zhao, "Word level symbolic model checking: A new approach for verifying arithmetic circuits," Pittsburgh, PA, USA, Tech. Rep., 1995.

[10] M. Ciesielski, P. Kalla, and S. Askar, "Taylor expansion diagrams: A canonical representation for verification of data flow designs," IEEE Transactions on Computers, vol. 55, no. 9, 2006, pp. 1188–1201.

[11] J. Møller and J. Lichtenberg, "Difference decision diagrams," Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, Aug. 1998.

[12] A. J. C. Bik and H. A. G. Wijshoff, Implementation of Fourier-Motzkin Elimination. Rijksuniversiteit Leiden. Valgroep Informatica, 1994.

[13] P. Bouyer, S. Haddad, and P.-A. Reynier, "Timed petri nets and timed automata: On the discriminating power of zeno sequences," Inf. Comput., vol. 206, no. 1, Jan. 2008, pp. 73–107.

[14] "Ask," [Retrieved: 16-October-2015]. [Online]. Available: http://www.ask-rfid.com/

[15] A. Ressouche, D. Gaffé, and V. Roy, "Modular compilation of a synchronous language," in Soft. Eng. Research, Management and Applications, best 17 paper selection of the SERA'08 conference, R. Lee, Ed., vol. 150. Prague: Springer-Verlag, August 2008, pp. 157–171.

[16] K. Brace, R. Rudell, and R. Bryant, "Efficient implementation of a bdd package," in Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE, June 1990, pp. 40–45.

[17] M. Abdelmoula, "Automatic test set generator with numeric constraints abstraction for embedded reactive systems," Ph.D. dissertation, Published in "Génération automatique de jeux de tests avec analyse symbolique des données pour les systèmes embarqués", Sophia Antipolis University, France, 2014.

Identifying Error-Prone Transactions in Enterprise Applications

Pavan Kumar Chittimalli, Sachin Patel, Vipul Shah
TCS Innovation Labs,
Tata Consultancy Services Limited,
Pune, India.
Email: {pavan.chittimalli, sachin.patel, v.shah}@tcs.com

***Abstract*—Independent testing teams use requirements as the basis to develop test cases and automated test scripts. The projects are executed under severe schedule constraints, due to which, the testers have to focus their testing efforts on error-prone and important features. Numerous source code based techniques for identifying error-prone features/components have been developed. However, they are based on source code analysis. Independent testing teams rarely have access to source code and they find it difficult to use code based techniques. In many cases, the domain experts use Business Process Model and Notation (BPMN) to represent the business requirements. In this paper, we propose an approach to identify error-prone transactions in enterprise applications using a BPMN. It helps in distinguishing between source code errors and test script errors. We have adapted this approach from an existing source code based technique. Our experiments with the approach show that it can identify the location of actual as well as seeded errors in both source code and test scripts.**

Keywords–Enterprise Application testing; BPMN; Stastical Bug Isolation; Bug Localization

I. Introduction

Business systems evolve due to various reasons such as correction of errors, adding new features, migrating to new environments, and improving performance. These changes may introduce infections [1], which propagate as the failure of the test-case. Testers face severe schedule constraints and they would like to spend their time on testing error-prone and important features. This necessitates the use of prioritization and fault localization techniques. There have been several fault localization techniques [1][2][3][4], proposed based on coverage information of the program entities and test executions logs. But teams which provide testing services do not have access to the source code. The testing team gets requirements in natural language or sometimes in formal notations like BPMN [5]. They use these as the basis to develop test cases and automated scripts [6]. In such scenarios, code based fault localization techniques cannot be used. The automated test scripts developed by testers are another source of error. It is difficult to differentiate between a test script failure and a source code failure. The manual trace analysis to identify test script errors, takes considerable amount of time and requires domain as well as technical expertise. This motivates us to develop techniques [7] for identifying error-prone features and test scripts of an application.

A. Motivating example

Listed below are two requirements of a billing application. The business transaction assumes to create an order and generate an invoice for it.

R1: If there *Exists Promotions* then apply the discount and generate invoice. If there *Exists No Promotion* then generate invoice without discount.

R2: In case of *Full Payment*, pay the generated invoice amount. In case of *Partial Payment*, pay amount less than generated invoice amount.

TABLE I. The sample test-scripts for the billing application

#	Test Sequence
T_1	1) Login 2) Create an order 3) If (promotions == 1) { apply discount generate invoice } 4) If (Payment Option==1) pay the amount 5) Logout
T_2	1) Login 2) Create an order 3) If (promotions == 1) { apply discount generate invoice } 4) If (Payment Option==2) pay the partial amount 5) Logout
T_3	1) Login 2) Create an order 3) If (promotions == 3) { generate invoice } 4) If (Payment Option==1) pay the amount 5) Logout
T_4	1) Login 2) Create an order 3) If (promotions == 3) { generate invoice } 4) If (Payment Option==2) pay the partial amount 5) Logout

The tester has identified four test cases from these requirements. See test case T_1 in Table 1. The first step is a *Login* with customer details like name and password. In the second step, *Create an Order*, displays the list of items to choose. The user selects items from the specified list and creates an order. In the third step, he checks for the option of any existing promotions (i.e., *promotions* $==$ *1*). The corresponding discounts are applied to the items ordered. Payment is done in the fourth step. If the payment option is *full payment* (i.e., *payment option* $==$ *1*) then pay the full amount and generate the invoice accordingly. The last step is a *Logout* event, which terminates the user session. Similarly, the other test-cases T_2, T_3, and T_4 are written as shown in Table 1 and executed with corresponding test-data. The execution results in successful execution (*pass*) for test-cases T_1 and failed execution (*fail*) for test-cases T_2, T_3, T_4. The reason for the failure of test-cases T_3 and T_4 is a script error at step 3. The script was checking for *promotions == 3* instead of *promotions == 2*, which resulted in a failure of the test-case. The test-case T_2 is failed because of source-code error in processing partial payment task.

Analyzing such errors requires lot of effort and domain and technical expertise. In this paper, we describe our technique of adopting the existing source code based fault localization techniques to a model based representation of the system - BPMN. In Section 2, we describe the approach for fault localization, followed by a description of two experimental studies in Section 3. We conclude the paper with a discussion of future work in Section 4.

II. Our Approach

We choose BPMN as a representation for functional requirements of the application. In this section, we first describe the BPMN using our sample example and the later part of the section will give the details of our approach based on this representation. Our approach tries to address the following research questions: 1) Can we adopt source code based fault localization techniques to BPMN model entities? 2) Can we distinguish between a script error and source code error?

The following subsections gives details about our approach.

A. Business Process Model

In BPMN terminology, a business process P is defined as: $P =< PE, F, s, E >$. The process element (PE) in BPMN representation can be a *task*, *gateway*, or a *subprocess*. A *task* is used for defining a particular activity. The *gateway* is used for decision making where each flow edge out of *gateway* has a condition associated with it. There are *or*, *nor*, *and*, *xor* variants of *gateway* exists as a representation. A *subprocess* is a place-holder or callee point for another business process. A *flow element* (an item of F) is an edge between two process elements. s is the *start* element. An *end* can be normal end of the process in which the return edge to callee exists. But in the *terminate end* the called process never returns to callee and ends the flow at that point.

For example, Figure 1 is a BPMN representation of the test cases shown in Table 1.

B. Test case, Test script generation using BPMN

A scenario (s_i) in the process diagram is defined as a path pa_i from start node s to end node e where $e \in E$. A path is a sequence of process elements (pe_i) with flow elements f_i in between each of those process elements defines a scenario s_i. Kholkar et al. [8] have proposed automating functional testing using a BPMN representation of the business application. We augment the standard BPMN representation with *pre* and *post* test conditions to specify test conditions and assertions. This results in a set of valid scenarios C for a process representation. For each such valid scenario, a test-case T_i is generated along with the scenario. For example, consider BPMN shown in Figure 1 for the illustration. The model has four feasible scenarios $\{s_1, s_2, s_3, s_4\}$ resulting in four unique test-cases $\{T_1, T_2, T_3, T_4\}$. The generated scenarios are shown in the following Table 2.

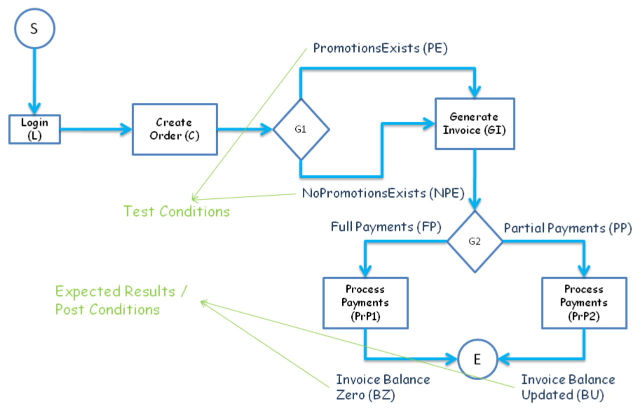


Figure 1. Annotate the billing application using functional requirements.

TABLE II. The scenarios for the example in Figure 1.

Id	Test case	Scenario
s_1	T_1	S → L → C → NPE → GI → FP → PrP1 → BZ → E
s_2	T_2	S → L → C → NPE → GI → PP → PrP2 → BU → E
s_3	T_3	S → L → C → PE → GI → FP → PrP1 → BZ → E
s_4	T_4	S → L → C → PE → GI → PP → PrP2 → BU → E

The test-case T_1 is generated for the scenario s_1 depicting a scenario - "*A registered user can create an order where there no promotions exists, and generate an invoice with full payment mode*". The same approach is described in our test automation tool [9]. This end-to-end script generation using BPMN representation of process diagrams are used in our approach for test-script generation.

C. Test execution and Traceability matrix building

The test automation tool in [9] is capable of capturing architectural, user interface, behavioral, and data models. This test automation tool records execution sequences at *entity* level for our process diagrams. The executions of these entities are then mapped to the corresponding test-cases using the execution traceability matrix. The empty cell in the traceability matrix indicates that the test-case does not execute the entity during the execution. The entry with a dark circle in the traceability matrix indicates that the entity has been executed during the test-case execution. The captured traceability matrix can be used in various regression testing and debugging activities [1][10][11]. The execution summary will result in either success or failure of the test-case. This execution status report (pass / fail information) and traceability matrix for the example in Figure 1 is shown in Table 3.

D. Identifying error-prone transactions

During the execution of test-cases on the system results in some failure and some successful executions. The cause of the failure can not be located by looking only into the failure test-cases [12]. In this subsection, we describe the adaptation of two source-code based fault localization techniques for use with BPMN. We first used Tarantula, a fault localization technique, invented by Jim Jones et al. [1][11][12]. Tarantula utilizes the pass / fail status of the test-case and the entities executed by each of the test-case. The other fault localization technique we used was Statistical Bug Isolation (SBI), a Liblit et al. [2]. In our approach, we adapted Tarantula, SBI and extended them to apply its metrics to entities in a BPMN model. Tarantula has two metrics *suspiciousness* and *confidence* to locate the error-prone entities in source code and *hue* and *brightness* to visually locate them. SBI has a metric called *Failure* to locate the error-prone entities in the source code. The *suspiciousness* of an entity *e* is defined as the level of being faulty that caused the failed test cases to fail. The value of suspiciousness metric ranges from '0' to '1', where '0', being least suspicious and '1' being high suspicious. Given a test-suite T, the suspiciousness metric for an entity *e* in Tarantula is defined as:

$$\begin{aligned} suspiciousness(e) &= \frac{\frac{failed(e)}{\#failed}}{\frac{passed(e)}{\#passed} + \frac{failed(e)}{\#failed}} \\ &= \frac{\%failed(e)}{\%passed(e) + \%failed(e)} \end{aligned} \tag{1}$$

In (1), $failed(e)$ represents the number of failed test-cases in T that have been executed by the entity e and $passed(e)$ represents the number of passed test-cases in T that have been executed by the entity e. $\#failed$ represents the total number failed test-cases and $\#passed$ represents the total number of passed test-cases in the test-suite T.The confidence metric is defined to state the confidence of the suspiciousness of the coverage *entity* that is being computed. The value of confidence ranges from '0' to '1' where '0' represents the least confidence and '1' represents the highest confidence, to the suspiciousness value. The *confidence* metric of entity *e* is defined as:

$$\begin{aligned} confidence(e) &= max\left(\frac{passed(e)}{\#passed}, \frac{failed(e)}{\#failed}\right) \\ &= max\left(\frac{\%passed(e)}{100}, \frac{\%failed(e)}{100}\right) \end{aligned} \tag{2}$$

In (2), the variables are same as in (1). The *max* takes the maximum value of fail/pass information available at that entity.

The $Failure$ of predicate *P* is defined as the probability of an atomic predicate (P) is true for failing runs and false for successful runs (i.e., $Pr(Crash|P\ observed\ to\ be\ true)$).

$$Failure(P) = \frac{F(P)}{S(P) + F(P)} \tag{3}$$

The $Failure(P)$ is expressed in the above equation where $S(P)$ denotes the number of successful runs in which *P* is observed true, and $F(P)$ denotes the failing runs in which P is observed to be true.

For example, consider in the process model defined in Figure 1. The test scenarios and test data are generated from annotated business process models [8]. The test script generation tool [13] is capable of capture and reply of the application. It records the coverage information of the entities in process model, which is used to build the traceability

TABLE III. TRACEABILITY MATRIX FOR THE TARANTULA TECHNIQUE

Entity Name	T_1	T_2	T_3	T_4	*suspiciousness*	*confidence*	*Failure*
$Start\ (S)$	●	●	●	●	.5	1	–
$Login\ (L)$	●	●	●	●	.5	1	–
$CreateOrder\ (C)$	●	●	●	●	.5	1	–
$gateway\ (G1)$	●	●	●	●	.5	1	–
$NoPromotionsExist(NPE)$			●	●	1	.6	.7
$PromotionsExist\ (PE)$	●	●			.25	1	.3
$GenerateInvoice\ (GI)$	●	●	●	●	.5	1	–
$gateway\ (G2)$	●	●	●	●	.5	1	–
$FullPayment\ (FP)$	●		●		.25	1	.3
$PartialPayment(PP)$		●		●	1	.6	.7
$ProcessPayment\ (PrP1)$	●		●		.25	1	–
$ProcessPayment\ (PrP2)$		●		●	1	.6	–
$BalanceZero\ (BZ)$	●		●		.25	1	–
$BalanceUpdate\ (BU)$		●		●	1	.6	–
$End\ (E)$	●	●	●	●	.5	1	–
Execution Status	**P** ✓	**F** ×	**F** ×	**F** ×			

matrix. The matrix is used to calculate the suspiciousness and confidence metrics. See the Table 3. In this case, test-case $\{T_1\}$ has passed whereas $\{T_2, T_3, T_4\}$ have failed. Using the coverage information and pass / fail information, the metrics *suspiciousness* and *confidence* have been computed. The entity *Start (S)* is executed by T_1 (pass) , T_2 (fail), T_3 (fail), T_4 (fail). Using the Tarantula approach, we calculated the corresponding *suspiciousness* for the *Start* entity as 0.5 and *confidence* as 1. Similarly, Using SBI approach, we computed the metrics for all other entities in the sample billing application. The rows 5, 10, 12, 14 in the Table 3 have the highest suspicious value '1'. For entity $NoPromotionsExist\ (NPE)$ (shown as suspicious in row 5) has been written wrongly in the test-cases T_3 and T_4 and categorized as test-script fault. The test-conditions $PartialPayments$ (in row 10) and $BalanceUpdate$ (in row 14) are with highest suspiciousness value (1) with a confidence value of 1. But the conditions do not have any faults so they are not classified as errors and $task\ processPayment\ (PrP2)$ (in row 12) is categorized as source code error in implementation of processing the payments. Similarly the rows 5 and 10 show the highest $Failure$ (SBI metric) as '.7' in failing predicates. The first predicate is failed due to test-script error and second failed due to source-code error in processing payments.

While the metrics help in identifying the error-prone BPMN entities, a visual representation would make it much easier to locate [12]. To achieve this feature, we used the color computing metric used in Tarantula. This technique uses Hue, Saturation, and Brightness (HSB) from *red* to *green* color range. We use *hue* metric to compute the color range specified in equations shown below. The *colorrange* is defined as 0.33.

$$\begin{aligned} hue(e) &= 1 - suspiciousness(e) \\ &= \frac{\%passed(e)}{\%passed(e) + \%failed(e)} \end{aligned} \tag{4}$$

$$color(e) = color(red) + hue(e) * colorrange \tag{5}$$

For example, see the Figure 2. The entity colored in *red* (*NoPromotionsExist*) is a highest suspicious entity. Analysis of the BPMN and test scripts reveal that it is a test script error. Instead of writing a test condition as $promotions == 2$, the condition has been mentioned as $promotions == 3$. The test-script and source code faults are shown in clouded area in Figure 2.

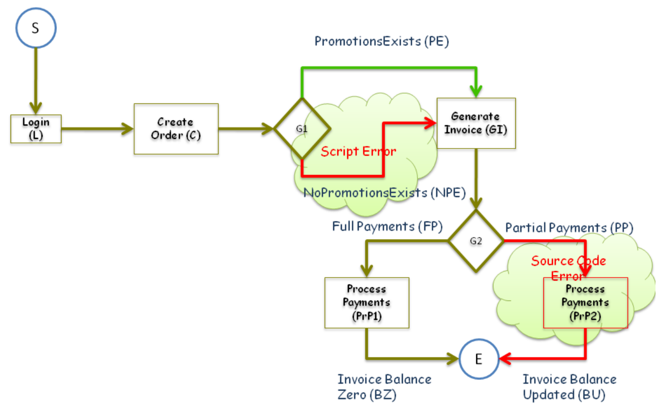


Figure 2. Using Tarantula visualization technique on billing application given in Figure 1.

The other suspicious entities shown in Figure 2 are *PartialPayments(PP)*, *InvoiceBalanceUpdated(IBU)*, *ProcessPayments(PrP2)*. On the careful observation of entities *PP* and *IBU*, we found that there is no script error exists. Hence the suspicious *task PrP2* is considered as source code error.

III. Experimental Studies

We evaluated the Tarantula [1] and Bin Liblit's approach [2] of SBI technique (Now, Co-operative Bug Isolation) along with our test generation toolset [8][13][9]. The systems under test were two open source enterprise applications, Jbilling and Mercury. Jbilling is an open-source billing system. We have configured it for a hardware construction material business. The process model for this application has 10 processes and 108 entities. 30 test-cases have been identified for Jbilling. The Mercury application is a online flight reservation system. The process model for this application has 3 processes and 29 entities. 13 test-cases have been identified for Mercury. The objective of our study is to 1) to locate parts of the business process that are error-prone 2) to distinguish between a source-code error and test-script error.

A. Study-1

We executed the identified test cases on the two applications. The results of the execution are shown in Table 4. In Table 4, the first column represents the subject. The second column shows the number of failures detected by Tarantula and SBI techniques. The third column shows the number of code failures detected by Tarantula and SBI. The fourth column shows the test-script failures detected by Tarantula and SBI respectively. The color coding provided by the tool helps locate the errors in source code, as well as test scripts. A red colored edge represents a test script error as this transition is caused by the test script and not the source code. If a fault is not test-script error then, we conclude it as a source-code error and point to the corresponding $task$ in the process model. We located one such fault in the Jbilling application. Consider the first row in Table 4 for Jbilling subject. Tarantula has detected 6 faults, of which, 5 failures are source code failures and 1 failure is a test-script failure. Similarly, SBI has detected the same for Jbilling application.

TABLE IV. The details of the Study-1 in experimentation.

Subject	Total Failures		Code Failures		Test-script Failures	
	Tarantula	SBI	Tarantula	SBI	Tarantula	SBI
$Jbilling$	6	6	1	1	5	5
$Mercury$	3	3	0	0	3	3

B. Study-2

In this study, we used seeded faults by generating more test scenarios. These additional seeded faults are created by mutating the operators in the process flow conditions of the process diagrams. The edges (Flow Elements) in BPMN are associated with the flow conditions. We have selectively taken these conditions for seeding. We applied operator mutation on relational operators $(>, <, \geq, \leq)$, equality operators $(=, \neq)$. We modified the test scenario generation described in section 2-B to address this. The objective of this study is to see if the mutants are killed or caught by the test scripts. We observed that all mutants have been caught as shown in the Table 5.

IV. Related Work

Most fault localization techniques in the literature have been based on code coverage. The common method has been to compare the coverage of failure runs and passing runs to determine the location of the faults. Jim Jones et al. [1][11][12] have done extensive research in the field of fault localization based on coverage of failure and passing runs. Their tool *Tarantula* uses the coverage information of entities at statement level to compute *suspiciousness*, *confidence*, *hue*, and *color* metrics. This tool is capable of showing the faults using visualization. But this tool was developed to locate source code faults. Liblit et al. [2] have proposed fault localization based on the coverage of predicates in failing and passing runs by sampling failure predicates. This is a lightweight technique as it uses very little program instrumentation compared to the *Tarantula* technique. Tarantula technique is more useful in in-house debugging whereas the SBI technique can be used in field debugging. Zeller et al. [4] have proposed a light weight instrumentation technique to capture the method call sequence coverage for locating the faults in java programs. Comparing the object-specific sequences predicts the defects better than just simply comparing the coverage. Naoya Maruyama and Satoshi Matsuoka [3] have proposed a fault localization technique in large computing systems using traces which capture function calls. They derive a model from the traces and compare them with failure traces to find the defect and computes suspect score to that failure.

In practice, the functional test teams carry out system and regression tests as independent test teams, treating the systems as a black box. Test teams prepare test plans and test scenarios from functional requirements that are available informally in natural language or sometimes semi-formally in notations like

TABLE V. THE DETAILS OF THE STUDY-2 IN EXPERIMENTATION.

Sub.	**No.of Test cases**	**Total Failures**		**Code Failures**		**Test-script Failures**		**Mutants Caught**	
		Tar	SBI	Tar	SBI	Tar	SBI	Tar	SBI
1	50	26	26	1	1	25	25	20	20
2	21	3	3	0	0	11	11	8	8

BPMN. Test scripts are manually or automatically generated from such models. The above fault localization techniques [1][11][12] that take into account coverage information of the program entities and test executions logs have been proposed. Independent test teams however do not have access to code nor the knowledge of the code to understand and interpret the results provided by current techniques. One of the additional challenges faced by the test teams, especially during the first test run in each release, is that the new test scripts may be faulty, or older test scripts may become out of sync with the requirements. A significant amount of time and effort is spent to determine if the faults are in the test scripts or source code. Further, with the advances happening in model-based testing, it is necessary to investigate if the code-based techniques developed so far have an utility in the model-based world. The work done in this paper is one such exploration.

V. CONCLUSIONS AND FUTURE WORK

The techniques applied in this paper have been extensively used with source code entities. We have applied them for a non-executable, model based representation. In this paper, we proposed BPMN as a system representation and extended the existing Tarantula, SBI techniques to identify error-prone transactions in an enterprise application. We also used Tarantula's visualization metric to locate faults in BPMN representation of the system.

Our preliminary experiments on in-house examples and openly available subjects showed encouraging results and caught all script and source code errors. These results have been manually verified. However, we have not applied this approach on a real-time project. A dependence fault, which appears only after fixing root faults, is not handled in current approach. Our assumption for locating source code fault has not verified in presence of dependence faults. We would like to apply this technique on bigger and more complex systems. Another possibility is to use test execution history to guide the test selection. Further studies will be required to understand the relationship between code-based and model-based metrics.

REFERENCES

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 15–26.

[3] N. Maruyama and S. Matsuoka, "Model-based fault localization in large-scale computing systems," in Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, April 2008, pp. 1–12.

[4] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in Proceedings of the 19th European Conference on Object-Oriented Programming, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 528–550.

[5] "Business Process Model And Notation (BPMN)," http://www.omg.org/spec/BPMN/, [Online; accessed 04-July-2015].

[6] Q. Yuan, J. Wu, C. Liu, and L. Zhang, "A model driven approach toward business process test case generation," in Web Site Evolution, 2008. WSE 2008. 10th International Symposium on, Oct 2008, pp. 41–44.

[7] P. K. Chittimalli and V. Shah, "Fault localization during system testing," in Proceedings of International Conference on Program Comprehension (ICPC), May 2015.

[8] D. Kholkar, N. Goenka, and P. Gupta, "Automating functional testing using business process flows," in Proceedings of Workshop on Advances in Model-Based Software Engineering, ser. ISEC (2011), 2011, pp. 102–110.

[9] S. Patel, P. Gupta, and P. Surve, "Testdrive - A cost effective way to create and maintain test scripts for web applications," in Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010, 2010, pp. 474–476.

[10] P. K. Chittimalli and M. J. Harrold, "Regression test selection on system requirements," in Proceedings of the 1st India Software Engineering Conference, ser. ISEC '08. New York, NY, USA: ACM, 2008, pp. 87–96.

[11] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in Proceedings of the 24th International Conference on Software Engineering, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[12] J. A. Jones, "Semi-automatic fault localization," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, Georgia, USA, April 2008.

[13] P. Gupta and P. Surve, "Model based approach to assist test case creation, execution, and maintenance for test automation," in Proceedings of the First International Workshop on End-to-End Test Script Engineering, ser. ETSE '11. New York, NY, USA: ACM, 2011, pp. 1–7.

Automatic Falsification of Java Assertions

Rafael Caballero   Manuel Montenegro
Universidad Complutense, Facultad de Informática
Madrid, Spain
email: {rafacr,mmontene}@ucm.es

Herbert Kuchen   Vincent von Hof
University of Münster, Institute of Information Systems
Münster, Germany
email: {kuchen,vincent.von.hof}@wi.uni-muenster.de

***Abstract*—We present an approach for the static detection of possible assertion violations in Java. The main idea is to use an existing test-case generator in combination with a new program transformation. A possible assertion violation is indicated by a generated specific test case. In addition, this test case specifies the path in the program leading to the assertion violation. This heuristic approach is a compromise between the usual but too late detection of an assertion violation at runtime and an often too expensive complete analysis based on a model checker.**

***Keywords*–assertion; automatic test-case generation; program transformation.**

I. Introduction

Assertions are part of the Java language [1] and have become part of the routine employed by Java programmers to detect and correct bugs. They can be used e.g. for specifying pre- and postconditions of methods or invariants of loops. If an assertion is violated, this is detected at runtime and a corresponding exception is thrown. A drawback is that it may take a long time, until assertion violations occurring in rarely executed code show up. Possibly, this can happen when the code has already been deployed and assertions are turned off [2, chapter 6]. Thus, the error can be difficult to detect and its correction become very costly.

Callahan et al. [3] proposed the use of model checkers for the automated generation of test cases. This model-based testing approach has been a fruitful area of research in the last years [4], and encompasses the creation of an abstract model which is used to automatically create test cases. However, using model checkers often requires more effort and expertise than the simple introduction of assertions. Additionally, the process of finding the assertion violation can still become a hard, time-consuming task due to the typically huge search space.

Our idea is to find some compromise between the two mentioned approaches and use a test-case generator [5], [6] to obtain test-cases for the considered code. If such a tool generates a test-case aiming at producing an assertion violation, this indicates that such an assertion violation can actually happen and that there is some corresponding bug in the program. Test-case generators do not explore the complete space of all possible computations as done by a model-checker. Typically, they apply a heuristic based on a combination of random search and symbolic evaluation in order to generate a set of test cases which cover the control- and/or data-flow of a program systematically [5], [7], [8], [9], [10], [11]. This approach cannot guarantee to find all possible assertion violations. Nevertheless, it works quite well and it is helpful in practice.

As mentioned before, a violation of a Java assertion causes an exception to be thrown. Unfortunately, test-case generators often have difficulties to cover exception handling well. Thus, our approach does not just rely on an existing test-case generator. Before using it, we apply a program transformation, which replaces assertions and the corresponding exception handling by "ordinary control structures". As we will show, this improves the coverage rate of test-case generators significantly. In addition, it allows test-case generators such as *jPET* [6] to be used, which do not support assertions.

Roughly, the approach presented here introduces new boolean methods representing the paths leading to possible assertion violations. In the case of methods including directly assertions, the body of the new method is a copy of the method where the assertion occurs, but replacing the assertion `assert` *e* by `return` *e*. This converts assertion violations into first-class citizens from the point of view of automated test-case generators, which usually focus on methods and their results.

The new `return` statements often produce fragments of unreachable code in the body of the new methods. These fragments can be automatically removed, thus simplifying the task of the test-case generator, and achieving a simple form of static slicing, as computations which are not relevant for assertions are not taken into account. This transformation is simpler than the alternative approach presented in [12], where every method is replaced by another one delivering a pair of the original result and a value indicating whether an assertion violation occurred. Propagating such violation information is technically a bit clumsy and the mentioned slicing is not obtained.

The paper is structured as follows. In Section II, we explain our transformation based on a running example, while in Section III, we present our transformation in detail. Section IV contains some experimental results. Finally, in Section V we summarize and point out future work.

II. Running Example

In order to get an overview of our transformation, let us consider the classes shown in Figure 1, which contain an implementation of the insertion-sort algorithm. An instance of `InsertionSort` contains a reference to the array to be sorted. This reference is initialised within the constructor, which previously checks, whether it is given a non-null reference. The `insert` method receives a number `n` and performs an ordered insertion of the element `x[n]` into the sub-array

```
public class InsertionSort {
   private int[] x;

   public InsertionSort(int[] x) {
      assert x != null;
      this.x = x; }

   public void insert(int n) {
      assert isSorted(n-1);
      assert n <= x.length;
      int i = n;
      while (i >= 1 && x[i-1] > x[i]) {
         int e = x[i-1];
         x[i-1] = x[i];
         x[i] = e;
      }
      assert isSorted(n); }

   public void insertSort() {
      for (int i = 1; i < x.length; i++)
         insert(i); }

   public boolean isSorted(int n) {
      for (int i = 1; i < n; i++)
         if (x[i-1] > x[i]) return false;
      return true; }
}

public class Check {
   public static void check(int []x) {
      InsertionSort is = new InsertionSort(x);
      is.insertSort(); }
}
```

Figure 1. Running example “insert sort”

x[0..n-1], which is assumed to be sorted. That is why we include an assertion that calls the method isSorted, which checks whether the array x is sorted up to the position given as parameter, disregarding the elements after that position. After the insertion, we check again (via another assert) that the resulting sub-array x[0..n] is sorted. Notice that there is a mistake in this method, as variable i should be decremented at the end of each iteration. Otherwise, the loop would always terminate either before the first iteration (if x[n-1] <= x[n]) or before the second one (when x[n-1] > x[n] holds before the first iteration, but not afterwards).

The insertionSort method calls insert as many times as the length of the array indicates, successively performing ordered insertions from the first element to the last one. Finally, class Check represents any arbitrary application class that employs an object of class InsertionSort. It is clear that some possible inputs of Check.check can trigger an assertion exception, exposing the existence of an error in the code. Our goal is to find such input values employing an automated test-case generator. In particular, it would be great, if the test-case generator could pay special attention to the assertions in the code, since any input data producing an assertion falsification reveals a code bug. This reduces the problem of checking the test-suite (known as the *oracle problem* [13]), as we can focus first on those test-cases producing assertion violations.

```
Input: A Java program $P$
Output: A Java program $P_0$ for the testing assertions

  $P_0 = P$
  for all method C.M $\in P$ containing assertions do
    Create a boolean copy C.M' of C.M in $P_0$
    Let $n$ be the number of assertions in C.M
    Let $a_j \equiv$ assert $e_j$ be these assertions, where $1 \leq j \leq n$, represents the textual order of occurrence of the assertion in C.M
    for $i = 1 \ldots n$ do
      Create in $P_0$ a new method C.M$_i^0$ as copy of C.M' except for:
      Assertion $a_i$ is replaced by return $e_i$
      Every $a_j \equiv$ assert $e_j$, $j < i$ is replaced by boolean $v_j = e_j$, with $v_j$ a new variable name
      Every assertion $a_j$ with $j > i$ is removed.
    end for
    Remove C.M'
  end for
```

Figure 2. Algorithm 1: Level 0, methods including assertions

However, some test-case generators, such as *jPET* [6] or *Muggl* [11], do not support assertions. Others, like *EvoSuite* [5] support assertions but have some problems when the assertions are located in a different class. In our running example, the three automated test-case generators find a test-case corresponding to the case of null array input, but fail to generate any test-case exposing the error in the code of method insert. In the next section, we present the program transformation that will change this situation.

III. TRANSFORMATION

Given a Java program including assertions, we introduce new methods that return the value false whenever the assertion property does not hold. Each method represents a certain path to an assertion violation.

In the rest of the section, given a method C.M we use the expression *create a boolean copy C.M' of method C.M* to indicate the creation a new method M' in class C such that C.M' is a copy of C.M except for:

1) The return type of C.M', which is boolean,
2) Statements return e; in C.M, which are replaced by return true; in C.M'.
3) If the type of C.M was void, then return true; is added as last statement of C.M'.
4) If C.M is a constructor, then add the access modifier static to the declaration of C.M'.

First, we produce the methods that correspond directly to methods containing assertions.

A. *Methods including assertions*

The first algorithm creates a transformed program P_0 as a copy of the initial P with some additional methods (see Figure 2).

Thus, a method C.M containing n assertions gives raise to n new methods C.M$_1^0$, ..., C.M$_n^0$, all of them with return type boolean and each one checking a particular assertion.

The auxiliary method C.M' is only introduced to facilitate the generation of the new methods, and is removed at the end.

For instance, in the case of the method `insert` of our running example, the C.M' method is obtained by replacing the return type by `boolean` and adding a new statement `return true;` at the end:

```
  public boolean insertPrime(int n) {
     // same body as insert
     ....
     return true; }
```

The method `insert` contains three assertions. Hence $n = 3$, and three new methods are included in the same class. The method associated to the first assertion is:

```
  public boolean insert1(int n) {
     return isSorted(n-1);// assertion
     int i = n;
     // code of the while loop in insert
     ....
     return true; }
```

Since the first statement is a `return`, any Java optimizer will prune the rest of the code, as it is unreachable, compiling instead:

```
  public boolean insert1(int n) {
     return isSorted(n-1); }
```

This is one of the main advantages of our approach: the new methods are often much smaller than the original ones and thus, the test-cases are obtained more easily. With respect to `insert_2`, we obtain:

```
public boolean insert_2(int n) {
   boolean _unused_1 = isSorted(n-1);
   return n <= x.length; }
```

Although we are interested only in the second assertion of `insert`, we still evaluate the condition of the first assertion, since it may involve side effects that may affect the result of the second one. The code of `insert_3`, the method associated to the last assertion, can be found in Figure 3.

It is worth observing that the constructors can be considered as any other method (except for the introduction of the `static` modifier), and no special treatment is needed. This is an important difference with respect to [12], where a more complicated treatment of constructors is necessary.

After this initial transformation, we can use our test-case generator to look for values $v_1, \ldots, v_k$ such that C.M$_i^0(v_1, \ldots, v_k)$ produces the value `false`, indicating that C.M$(v_1, \ldots, v_k)$ triggers assertion a_i. However, we would like to go one step beyond and consider if such a call can actually occur in our application. This is the purpose of the algorithm in the following subsection.

B. Indirect access to assertions

We say that the level of indirection of a method C.M is zero, if the method contains an assertion (case considered in the previous section), and $l > 0$, when it contains a call to method C'.M', and C'.M' has a level of indirection $l - 1 \geq 0$. The idea behind this definition is that methods with levels greater than zero can end triggering an assertion and must be transformed as well. If a method does not contain an assertion and it does not contain method calls (possibly indirectly) leading to assertions, the level of indirection is undefined.

Notice that the same method can have different levels of indirection related to different method calls. For instance, method `Check.check` (Figure 1) has an indirection level of 1 with respect to the call of constructor `InsertionSort` that contains an assertion, but also level 2 with respect to the call `is.insertSort`, as method `InsertionSort.insertSort` has indirection level 1. A *maximal level of indirection* is assumed as an input parameter of the following algorithm. It can either be obtained previously by the tool analyzing the code of considered as an input parameter fixed by the user to limit the number of methods generated in case of large applications.

Our transformation creates new methods associated to each level of indirection. We assume that it is possible to distinguish the auxiliary methods created when processing a method at a certain level of indirection. For instance, all the auxiliary methods C.M$_i^0$ created by Algorithm 2 correspond to the transformation at level 0 of their source method C.M.

Now, we apply the transformation to the `insertionSort` method, which does not contain any explicit assertion, but it may indirectly trigger an assertion violation in `insert`. The call to `insert` occurs in a loop, and hence first the loop is unfolded. Let us suppose that the parameter *Unfold* takes value 2.

```
public void insertSortPrime() {
  int i=1;
  if (i<x.length) {
     insert(i);
     i++;
     if (i<x.length)
        insert(i); } }
```

In the algorithm, this means that $p = 2$ (two calls to `insert`) and $q = 3$ (three versions of `insert` have been already generated). Thus, we obtain a family of $2 \times 3 = 6$ methods `insertSort_`i`_`j, with $1 \leq i \leq 2$ and $1 \leq j \leq 3$. The method `insertSort_`i`_`j checks whether the condition of the j-th assertion executed within the i-th call of `insert` is satisfied.

For instance, `insertSort_1_3` in Figure 3 corresponds to the possibility that the first `insert` in `insertSortPrime` falsifies the post-condition of `insert` (its third assertion).

The transformation can also be applied to constructors, like `InsertionSort_1` prefix. Therefore, when transforming a method that involves the instance creations of `InsertSort`, we have to replace the `new InsertSort(...)` expressions by calls to this static method in order to check the validity of the assertions contained in the constructor. This is the case of `Check.check_1_1` in Figure 3.

The same figure includes a family of methods `check_2_`i`_`j that reports the assertions being violated by the indirect through the `insertSort` method. Each

```
.... // original methods

public static boolean
   InsertionSort_1(int[] x) {
   return x != null; }

public boolean insert_1(int n) {
   return isSorted(n-1); }

public boolean insert_2(int n) {
   boolean _unused_1 = isSorted(n-1);
   return n <= x.length; }

public boolean insert_3(int n) {
  boolean _unused_1 = isSorted(n-1);
  boolean _unused_2 = n <= x.length;
  int i = n;
  while (i >= 1 && x[i-1] > x[i]) {
     int e = x[i-1];
     x[i-1] = x[i];
     x[i] = e;
  }
  return isSorted(n); }

 public boolean insertSort_1_1() {
   int i = 1;
   if (i < x.length)
      return insert_1(i);
   return true;}

 public boolean insertSort_1_2() {
   int i = 1;
   if (i < x.length)
      return insert_2(i);
   return true; }

 public boolean insertSort_1_3() {
   int i = 1;
   if (!(i < x.length))
      return insert_3(i);
   return true;}

  public boolean insertSort_2_1() {
    int i = 1;
     if (i < x.length) {
        insert(i);
        i++;
        if (i < x.length)
          return insert_1(i);}
    return true;}
```

```
public boolean insertSort_2_2() {
 int i = 1;
 if (i < x.length) {
    insert(i);
    i++;
    if (i < x.length)
      return insert_2(i);}
 return true; }

public boolean insertSort_2_3() {
 int i = 1;
 if (i < x.length) {
    insert(i);
    i++;
    if (i < x.length)
       return insert_3(i);}
 return true;}


public class Check {
 public static boolean check_1_1(int []x) {
  return InsertionSort.InsertionSort_1(x);}

 public static boolean check_2_1_1(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_1_1(); }

 public static boolean check_2_1_2(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_1_2();}

 public static boolean check_2_1_3(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_1_3();}

 public static boolean check_2_2_1(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_2_1();}

 public static boolean check_2_2_2(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_2_2();}

 public static boolean check_2_2_3(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_2_3();}}
```

Figure 3. Running example after the transformation

of these methods subsequently calls the corresponding `insertSort_`i`_`j variant.

After the transformation and in order to check, whether an assertion may be violated at runtime, we just have to invoke a test-case generator on one of these generated methods and look for those cases that yield `false` as a result. For instance, when given the method `check_2_2_3`), *jPET* generates a test case (an instance of `InsertionSort` containing the array $[-8, -9, -10]$) which violates the third assertion executed by the second call to `insert`. Analogously, *EvoSuite* and *Muggl* also find an assertion violation associated with `check_2_2_3`. Observe that the name of the method specifies a very detailed scenario: it indicates that with the given input array, `check` causes an assertion falsification in its sec-

```
Input:
    • Program $P_0$: output of Algorithm 1
    • Level: maximum level of indirection allowed (greater than 0).
    • Unfold: A positive number indicating the number of iterations to unfold the loops in the body methods.
Output: $P^T$: A Java program ready to be used to obtain the test-cases falsifying the properties indicated in the assertions.

  $P^T = P^0$
  Mark $P^T$ methods containing assertions with level 0.
  for l=1 ...level do
    for all method D.N in $P^T$ containing calls to methods marked as level $l-1$ do
      Mark D.N as method of level $l$.
      Create a boolean copy D.N' of D.N in $P^T$
      if any call to a $l-1$ method in D.N' occurs in a loop statement then
        Unfold the loop in the copy D.N' the number of times specified by the algorithm input parameter unfold.
      end if
      for all call T x = C.M(...); occurring in method D.N', with C.M marked as level $l-1$ do
        Let $p$ be the number of calls to method C.M in D.N'
        Let C.M$^{l-1}_{s_1}$, ..., C.M$^{l-1}_{s_q}$ be the $q$ auxiliary methods created at level $l-1$ for C.M
        for i=1 ...p, j = 1 ...q do
          Create a copy D.N$^{l}_{i,s_j}$ of D.N'.
          Replace in D.N$^{l}_{i,s_j}$ the statement T x = C.M; by return C.M$^{l-1}_{s_1}$;.
        end for
        Delete D.N'
      end for
    end for
  end for
```

Figure 4. Algorithm 2: Level of indirection greater than 0

ond call `is.insertSort();` (first number 2). Moreover, it also shows that `insertSort` causes the assertion falsification in the second iteration of the loop (this is represented by the second number 2 in the name), and the falsification occurs in the last assertion of `insert` (final number 3).

By including the decrement instruction `i--;` at the end of the loop within `insert` (as explained above), no assertion violations are found.

IV. Experimental Results

To observe the effects of the transformation, we have utilized experimentation. In addition to the running example shown above, we have investigated several additional examples [14], ranging from the implementation of the binary tree data structure, Kruskal's algorithm, to Mergesort. Finally, we used two examples representing a blood donation scenario *BloodDonor* and a larger application, namely a library system, where users can lend and return books. In the next step, we have evaluated the examples with different test-case generators with and without our program transformation.

TABLE I. Detecting assertion violations

		EvoSuite		jPet	
Example	Total	P	P^T	P	P^T
InsertionSort	4	3	4	0	4
CircleRadius	2	2	2	0	2
BloodDonor	2	1	2	0	2
InsertTree	2	1	2	0	2
Kruskal	1	1	1	0	1
Library	5	0	5	0	5
MergeSort	2	1	1	0	1
Numeric	2	2	2	0	2

We have used two test-case generators for exposing possible assertion violations. First of all, we can note that this approach works. Moreover, we can note that our program transformation typically improves the detection rate, as can be seen in Table I. In this table, column *Total* displays for each example the number of possible assertion violations that can be raised for the method. Column P shows the number of detected assertion violations using the test-case generator (0 in the case of JPet because it does not handle assertions) and the original program and column P^T displays them after applying the transformation. For instance in our running example four assertion violations can be raised. Without the transformation, three assertion violations are found by EvoSuite. With the transformation, EvoSuite correctly detects all four assertion violations. An improvement in the assertion violation detection rate is observed for all examples. *jPET* does not consider assertions in its current state, but can detect them after our program transformation.

Thus both tools that do and do not support assertions benefit from our program transformation. The runtime of our analysis can range from a few seconds to several minutes.

V. Conclusion

Assertions constitute a useful, widely-used feature of the Java language. They are widely used for detecting bugs in the testing-phase. However, only those assertion violations actually occurring at runtime can be detected.

Automated test-case generators can be situated somehow in the middle of the very light-weight technique of run-time checking Java assertions and the formal methods such as model checking. They do not require the definition of abstract models, but aim to cover as many executions as possible of the program, yielding test-case suites that can be used to look for possible errors. The main difficulty is to check the generated test-suites looking for test-cases producing erroneous results. This is known as the *oracle problem* [13]. In order to solve this problem, [15] proposes including the assertions as part of the code and use automated test-case generation to obtain inputs that falsify the conditions. This approach was already presented in [11] and has given rise to the so called assertion-based software-testing technique.

In this paper, we have presented a proposal for transforming a Java program including new boolean methods that help to check the program assertions. Each of these methods returns false, whenever its input parameters lead -directly or indirectly- to a falsification of some assertion property. Moreover, the name of the method contains a path to the assertion.

Some automated test-case generation tools do not consider assertions. The presented transformation allows the user to employ even such test-case generators to generate test-cases exposing assertion violations. Moreover, we have seen that it can also contribute to increase the completeness of the test-cases obtained in some tools such as *EvoSuite* [5] that already consider assertions. We also think that this proposal can be useful during the development of new test-case generators in order to include readily the possibility of dealing with assertions. The advantage of our technique is that assertions are replaced by standard code that can be analyzed using the usual techniques.

It is worth observing that using our transformation, the test-cases corresponding to assertions are easy to distinguish, since they correspond to new auxiliary methods returning `false`. Thus, it is possible to implement readily an automatic tool that extracts from the test-suite the test-cases falsifying assertions.

The main limitations of the proposal are:

1) The necessity of unfolding the loop statements where assertions are included. Since the unfolding is done a fixed number of times, this can reduce the effective covering of the test-cases.
2) The combinatorial explosion in the number of methods. We have seen in the description of the transformation that if a method contains n assertions and is called m times by other methods, we need to generate n auxiliary methods for the first one and $n \times m$ auxiliary methods for the second one.

The unfolding (or 'unrolling') of the loops containing methods using assertions is not a very severe restriction in practice, because most automated test-case generators do the same internally. Moreover, we have found that most errors show up after just two iterations, like in the running example of this paper. Anyway they can add more incompleteness to the results.

The positive part of unfolding the loops is that errors found are very precise. In our running example we can check that all the methods leading to assertion violations require two iterations of the loop. This points out the updating of variables at the end of the loop as a possible cause of the bug, which is indeed the case. To the best of our knowledge, no test-case generator can provide such detailed information.

The combinatorial explosion in the number of auxiliary methods can become an issue for large programs with many assertions. We have found that processing each assertion separately instead of all the assertions at the same time results in a considerable speed-up. Anyway, it is worth observing that the process is automatic and requires no user-interaction once it has been started.

As future work, we plan to finish a prototype that automatizes both the transformation and its connection with different test-case generators. An important part of the prototype is the decodification of the auxiliary method names once an assertion falsification has been found, in order to show to the user a detailed information about the source of the bug. We also plan to extend the framework to the case of inheritance and polymorphism. Our preliminary results in this sense indicate that the same technique can be applied in the presence of polymorphism with the creation of 'dummy' auxiliary methods in the ancestor classes of the class hierarchy to ensure that the method exists and can be used also in polymorphic contexts.

ACKNOWLEDGMENT

This work has been supported by the German Academic Exchange Service (DAAD, 2014 Competitive call Ref. 57049954), the Spanish MINECO project CAVI-ART (TIN2013-44742-C4-3-R), Madrid regional project N-GREENS Software-CM (S2013/ICE-2731) and UCM grant GR3/14-910502.

REFERENCES

[1] Oracle, "Programming With Assertions," http://docs.oracle.com/javase/6/docs/technotes/guides/language/assert.html, retrieved: August, 2015.

[2] G. Travis, JDK 1.4 Tutorial. Manning Publications, 2002.

[3] J. Callahan, F. Schneider, and S. Easterbrook, Eds., Automated software testing using model-checking, 1996, proceedings 2nd SPIN workshop.

[4] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," Int. J. Softw. Tools Technol. Transf., vol. 17, no. 1, Feb. 2015, pp. 59–76. [Online]. Available: http://dx.doi.org/10.1007/s10009-013-0291-0

[5] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025179

[6] E. Albert, I. Cabanas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutierrez, "jPET: An automatic test-case generator for Java," in 18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011, 2011, pp. 441–442.

[7] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in IEEE International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2013, pp. 360–369.

[8] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[9] M. Gómez-Zamalloa, E. Albert, and G. Puebla, "Test case generation for object-oriented imperative languages in CLP," TPLP, vol. 10, no. 4-6, 2010, pp. 659–674. [Online]. Available: http://dx.doi.org/10.1017/S1471068410000347

[10] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[11] M. Ernsting, T. A. Majchrzak, and H. Kuchen, "Dynamic solution of linear constraints for test case generation," in Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China, 2012, pp. 271–274. [Online]. Available: http://dx.doi.org/10.1109/TASE.2012.39

[12] R. Caballero, M. Montenegro, H. Kuchen, and V. von Hof, "Checking java assertions using automated test-case generation," in 25th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2015), 2015, retrieved: August, 2015. [Online]. Available: https://gpd.sip.ucm.es/rafa/papers/lopstr15.pdf

[13] E. Barr, M. Harman, P. McMinn, M. Shabaz, and S. Yoo, "The oracle problem in software testing: A survey," IEEE Transactions on Software Engineering, vol. PP, no. 99, 2014, pp. 1–1.

[14] R. Caballero, M. Montenegro, H. Kuchen, and V. von Hof, "Examples used," https://github.com/wwu-ucm/valid-15-examples, retrieved: August, 2015.

[15] B. Korel and A. M. Al-Yami, "Assertion-oriented automated test data generation," in Proceedings of the 18th International Conference on Software Engineering, ser. ICSE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 71–80.

RDTA – Repository Driven Test Automation

A new look into reuse of test automation artifacts

Dani Almog, Hadas Schwartz Chassidim, and Shlomo Mark
Dept. of Software Engineering
Sami Shamoon College
Beer Sheva, Israel
e-mail: Almog.dani@gmail.com, hadasch@sce.ac.il, marks@sce.ac.il

Yaron Tsubery
R&D Operations
Enghouse Interactive
9th Nehar Prat St., Giva'at-Ze'ev, Israel
yaron.tsubery@gmail.com

Abstract— Repository Driven Test Automation (RDTA) is an approach to the buildup process of test automation infrastructure which proposes reuse of testing artifacts as a fundamental principle for the creation of test automation. Our research was motivated by a two-fold inquiry: Can testing automation artifacts be reused? If so, how? These inquiries led us to a new concept for the formulation of test automation. The term software repository here refers to a storage location from which software packages or artifacts may be retrieved for reuse in other systems or software products, preferably - as is. This conceptual paper explores different aspects of the reuse of software test automation artifacts and elaborates on several practical implications and changes that arise from the implementation of this new paradigm in a software development organization.

Keywords-testing; test automation; software reuse; repository driven automation.

I. INTRODUCTION

Testing is perhaps the most expensive task in a software project. Large portions of testing costs are derived from the need to assure that none of the newly introduced changes in the code have damaged previous quality – testing for regression is a repetitive activity. Regression testing is an expensive activity that can account for a large proportion of the software maintenance budget [1]. Software engineers add tests into test suites as software evolves, and by this increase the test suite size, the revalidation of the software but, also the testing costs. Special techniques to reduce the regression tests costs by selecting, prioritizing and reducing the number of regression tests and costs, have been proposed [1,2]. However, it can be expensive to employ these techniques and therefore it might not reduce the overall regression testing costs. A survey of practitioners [2] shows that the main benefits of test automation are: reusability, repeatability and effort saved in test executions. Automation can be applied to parts of the testing processes by entrusting repetitive tasks to a test automation system. The main motivation of RDTA is to reduce the overall expenses and efforts in the implementation of test automation by addressing test automation artifacts and the creation process itself [3]. Today, many commercial and open source tools are used for test automation. Large portions of these tools are highly specialized solutions for specific aspects of testing, are focused on different technologies, or are based on particular test paradigms. There is a large variety of specialized test tools for test case generation, test management, test execution, and so forth. There is limited support for combining the numerous specialized tools in an integrated solution except for the provision of technical interfaces between single tools.

The objective of our work is the development of test automation infrastructure rooted in the concept of reusing testing artifacts. In Section II, we briefly revisit the general reuse concepts, including some heuristics [4], and elaborating on some needed architectures, and testing artifacts and other dimensions of test automation. RDTA is introduced in Section III, discussing what, where and how to store the different artifacts. We conclude with conceptual insights into the implications of RDTA in today's modern software development arena (e.g., Unit test, agile, integration, Service-Oriented Architecture (SOA)).

II. BACKGROUND: REUSE OF ARTIFACTS

Analyzing our day-to-day testing activities, we may ask: how much of every action, operation, thinking, doing – is actually uniquely new? When attempting to explain the nature of the reusability concept, we may be challenged by the argument that this has all been done before and, therefore, that there is nothing new to contribute in this field. These notions are almost right: most new contribution stems from context and interpretation. For example, when designing a new test case for a certain application, memory and past experience are utilized to rearrange old knowledge into a new pattern to create a new test case that ought to answer the new aspects we are testing. So from a conceptual standpoint, we are reusing. In this paper we will examine how much reuse is done with regard to testing artifacts. In addition, we review the extent to which we are aware of the reusable nature of our work when designing a testing artifact. Our day-to-day manual testing work flow is built out of |context| –> |concept| –> |build| –> |use|. We will also review how much of a software engineer's attention/awareness is focused on the issue of reuse [5].

The reuse of artifacts is usually derived from the desire to take advantage of previously developed components and capabilities. In previous scholarship [6], a distinction was made between Development with Reuse (DWR), which focuses on benefits gained from the utilization of reusable resources, and Development for Reuse (DFR), which aims at the creation of reusable products for the benefit of future usage. A generalized reuse model for system development was formulated, suggesting a future quantitative evaluation of reuse in a comprehensive manner [6].

A. Reuse Heuristics

Fortue and Valerdi [4] Addressing the topic of reuse from a systems engineering perspective, a generalized framework for the reuse of systems engineering products has been proposed. This approach is based on reuse heuristics (the following is a partial list selected from the original study) [7]:

- Heuristic a: Reuse is not free, upfront investment is required.
- Heuristic b: Reuse should be planed from the conceptualization phase of programs.
- Heuristic c: Most project related products can be reused.
- Heuristic d: Reuse, in large part, is also an organizational issue.
- Heuristic e: Higher reuse opportunities exist when there is a match between the diversity and volatility of a product line and its associated supply chain.
- Heuristic f: Bottom-up (individual elements where make or buy decisions are made) and top-down (where product line reuse is made) reuse require fundamentally different strategies.
- Heuristic g: Reuse applicability is often time dependent.
- Heuristic h: The economic benefits of reuse can be described in terms of either improvement (in quality, risk identification) or reduction (of defects, cost/effort, and time to market).

The ability to recompose reusable parts is an important requirement for reuse [8]. Anticipating future reuse scenarios make reusable parts easier to compose. Khusidman and Bridgeland [9] presented a framework of reuse and cloning techniques in software development. This work analyzed different aspects of reuse and cloning by utilizing a classification framework to define a matrix of reuse scenarios aimed at efficient reuse. A distinction may be made between "formal" reuse of object code that does not require any customization, and the "opportunistic" "cut-and-paste" reuse achieved by using and modifying fragments of existing solutions [9]. In the following sections, we will attempt to generalize a reuse framework and apply its principles to test automation.

B. Systems Reuse Framework

It has been said that "reuse can increase your productivity by nearly half if you avoid the common pitfalls that derail many reuse programs" [10]. This idea was made clear from the analysis of the outcome of trends in Source Lines of Code (SLOC) of Department of Defense (DoD) software and DoD cost in dollars per SLOC between 1950 and 2000 [10].

However, reuse in software development and testing may present some abuse dangers, such as the propagation of errors in subsequent versions of the software [11]. Lengthy research on reuse of a test case in a safety critical system (for a heart pacemaker) [12] concluded that, conceptually, this approach to reuse is simple, but to implement it in a real project with hundreds of thousands of lines of code, recognizing the commonalities among the test cases, and implementing a mechanism for systematic reuse, is a huge task. Applying reuse techniques at the testing stage of a real project that involved the development of a cardiac rhythm management system led to significantly reduced efforts required to test systems. More recent studies relates reusability to Software Product Line Testing (SPLT) [13] [14] [15]. The strategy of reuse of core assets in SPLT can reduce software testing efforts during development, improve software quality, and potentially decrease the time-to-market of products and services.

C. Reuse of Testing Artifacts

Tiwari and Goel [16], the authors of a wide survey of the literature about the reduction of testing effort through reuse have argued that although there are many systematic studies that deal with quality assurance techniques, virtually no literature or survey exists on reuse-oriented testing approaches. RDTA deals with the reuse of testing automation artifacts using a comprehensive multi-level reuse approach.

III. RDTA: A Novel Approach

In this section, we present our contribution to the reuse classification framework by laying out the organizational structure for the different levels, types and candidates of storage repositories. The classification system presented here is preliminary and may be further expanded and adapted to different technologies and software development infrastructures.

Even before we consider test automation, we are obligated to store and maintain the testing artifacts. In addition to the Gupta test repository classifications [17], previous research and papers provide a useful resource for testing item classifications, for example, verification items [18]. Complimentary to the Gupta classifications, and in accordance to the research of most of experts in the test automation industry, we propose this preliminary list of subjects be stored:

- Requirements Repository (a)
- Business Story Repository (b)
- Test Cases Storage (c)
- Basic Operational Element Repository (d)
- Unit Test Repository (e)
- Testing Business Element Repository (f)

Other repositories may be introduced as the result of this research project. All these repositories support reuse when transmitted to, and prorogated among, organizations and teams.

A. *Business and Testing Requirements*

The RDTA approach suggests the requirement for a source link for most testing artifacts. It RDTA samples a subset of configurations to be tested based on environment modeling, requirement analysis and systematic traceability.

RDTA distinguishes between higher Business requirements and their breakdown into functional requirements and nonfunctional requirements.

Therefore, it is imperative that there be a depository where the entirety of the testing requirements are stored, maintained and controlled. Additionally, today many test management tools contain their own storage of test requirements and are linked and traceable to the software requirements as well as to the rest of the testing artifacts.

B. *Business Story Repository*

Derived directly from the store of software testing requirements is a depository of testing business stories. These should be as tightly aggregated as possible. Different aggregation levels may be represented in a repository for these testing stories or business story fragments. For example, "The customer should be able to access the application from most popular interfaces (mobile, pc, remote interface etc.) using a login procedure".

C. *Test Cases Storage*

The test cases repository should derive from the test business stories depository. Please note that test cases are very much application/functionality oriented and therefore

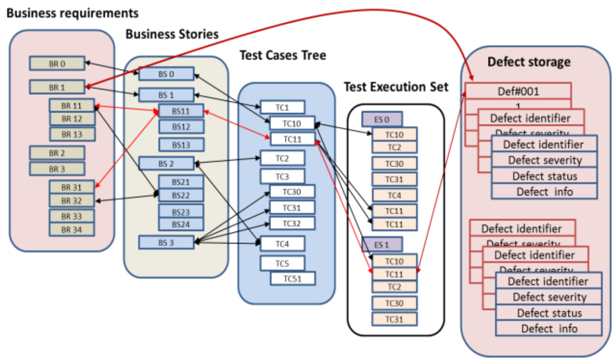


Figure 1. suggested test coverage metrix.

require storage in different hierarchies that allow for different affiliations or relationships to be exposed and identified. Figure 1 presents a possible traceability matrix that demonstrates the need for documentation as well as management and control at all items during the testing/fixing operation. Each column presents repository categories containing other testing artifacts. The arrows hint to a possible dependency between the elements.

These types of coverage matrices enable tractability [15], and may help may reveal the importance of keeping track of, and documenting, all business and testing artifacts.

D. *Basic Operational Element Repository*

In order to facilitate test automation needs, we must be able to execute and operate all developed applications under conditions of control and isolation. This can be performed during the development phase or in an integration workplace until installation. RDTA divides these repositories into two categories:

1) Operational infrastructure, architectural foundation related storage, and application.

2) Business related storage.

The reusable quality of the items stems from the similarity in the basic application of the actual business behavior in the software.

Each of these artifacts may be used, operated, stored, maintained and manipulated during the testing project. More items may be added and specifically modified. The use of these artifacts is limited by resource constraints and time horizons.

E. *Unit Test Repository*

In order to maintain productive reuse of unit test artifacts, isolated and single purpose (used mostly by developers) unit tests need to be transformed into integrated parts of reusable testing artifacts that are used by all levels of development and quality assurance teams [19].

F. *Testing Business Element Repository*

The need for the reuse of the same generic test case as part of a project scenario that has a different categorical affiliation can be satisfied in most of the existing testing

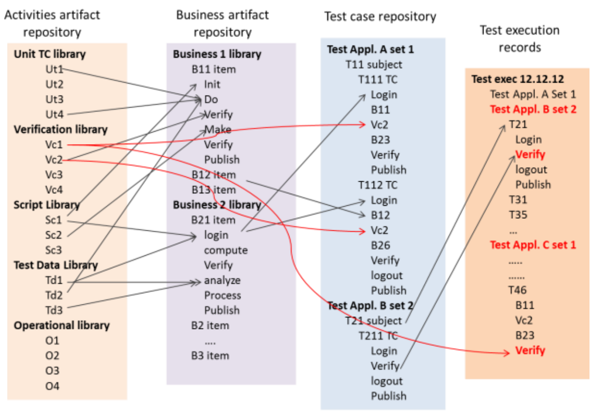


Figure 2. Principal RDTA testing repositories build up

tools by duplicating the same formation and storing it separately. The RDTA approach will store another level of artifacts that relate to the test case business context (see Figure 2).

To facilitate easy access and usage of reusable testing artifacts, the RDTA approach mandates adding another merged level: one that stores, uses and maintains another practical set of testing items. Business artifacts may be related to each element (or object) of the testing artifacts that can be treated as a business portion (as opposed to technical, architectural or other such element).

RDTA will recommend storing and maintaining the actual full context testing scripts so that during subsequent use, the user will have full control of all operational and functional aspects to be tested.

G. *Maintaining the Integrity of the Specifications*

The RTDA approach suggests a framework where each of the elements is categorized as a service – so it can be recalled and operated independently during the progression of the testing levels. Such a complex, interconnected, and affiliated storage system must be formulated in a very practical manner. Therefore, how and where to store objects are critical issues.

H. *How to Store Repositories*

Reflecting on the operational practical needs for the storage requirements, the following list of storage requirements has yet to be fully researched and evaluated:

- Easy & efficient storage & retrieval (Ease of use)
- Support for all types of items (from single data items to complex executable modules)
- Support for version control
- Ability to follow complex associations between the items
- Support for dynamic hierarchy relationships
- Discoverable and presentable on multiple layers and dimensions
- Easy to maintain
- Ability to follow security requirements
- Unlimited size.

I. *RDTA and the Test Automation Creation Work Process*

Adapting the RDTA approach mandates a new four step work process.

1. Analysis of project artifacts and the creation of a project repository.
2. Mapping the affiliations of project artifacts to existing reusable artifacts.
3. Acquiring test artifacts from the common repository for insertion into the project repository.
4. Designing missing test artifacts at the project repository and operation of automatic upload of the new test artifacts to the common repository.

J. *Implementing RDTA*

Implementing RDTA may prove to be a hard and complicated task in light of the variability and complexity of infrastructure, organizational cultures, standards and new quality measurements. One can foresee two different approaches for implementation:

- Top to bottom – where management dictates, supervises and imposes changes in production.
- Bottom up –where change develops from the bottom through limited experimental trials of one of the test automation teams and subsequently percolates up and spreads gradually through the organization.

IV. CONCLUSION

This paper presents a new conceptual approach to test automation – RDTA. This approach focuses on the reuse principle for test automation artifacts. In order to transition from concept to practice, each subject and proposition presented here should be addressed and developed into an organizational strategy and framework to reduce costs. More broadly, we envision the creation of international sharing schemes for the purpose of resource and performance amplification. Further development of the criteria for the selection of services and the evaluation of RDTA benefits are required.

REFERENCES

[1] , A. G. Malishevsky, G. Rothermel and S. Elbaum, "Modeling the cost-benefits tradeoffs for regression testing techniques". Software Maintenance. Proc. International Conference on, IEEE, pp 204-213, 2002.

[2] D. M. Rafi,. K. R. K. Moses, K. Petersen, and M.V. Mäntylä,."Benefits and limitations of automated software testing: Systematic literature review and practitioner survey". Proc. of the 7th International Workshop on Automation of Software Test, IEEE Press, pp. 36-42, June 2012.

[3] D. Almog and Y. Tsubery, "How the Repository Driven Test Automation (RDTA) will make test automation more efficient, easier & maintainable". Proceedings of the 8th India Software Engineering Conference. Bangalore, India, ACM: 196-197, Feb 2015.

[4] J. Fortue and R. Valerdi,, "A Framework for Reusing Systems Engineering Products," Syst. Eng. vol. 16, no. 3, pp 304-312, 2013.

[5] J. Parsons and C. Saunders, "Cognitive Heuristics in Software Engineering Applying and Extending Anchoring and Adjustment to Artifact Reuse," IEEE Trans. Softw. Eng., vol. 30, no. 12, pp. 873-888, Dec 2012.

[6] G. Wang and J. Rice, "Considerations for a Generalized Reuse Framework for System Development." Proc. 21st INCOSE Int. Symp., June 2011.

[7] C. E. Cagdas, K. Bhattacharya, J. Su, "Static Analysis of Business Artifact-centric Operational Models," 2007 IEEE Int. Conf. on Serv. Oriented Comput. and Appl., June 2007, pp. 133-140.

[8] A. H. Bagge, M. Bravenboer, K. T. Kalleberg, K. Muilwijk, E. Visser, “Adaptive Code Reuse by Aspects, Cloning and Renaming,” Tech. Rep. UU-CS, issue: 2005-031 (2005).

[9] V. Khusidman and D. M. Bridgeland: “A Classification Framework for Software Reuse”, J. of Object Technol., vol. 5, no. 6, pp. 43-61, July - August 2006.

[10] B. Boehm,. “Managing Software Productivity and Reuse,” Computer, vol. 32, no. 9, pp. 111-113 1999.

[11] E. J. Weyuker, “Testing Component-Based Software: A Cautionary Tale”. IEEE Softw., Vol. 15, No. 5: pp. 54-59, 1998.

[12] M. Poonawala, S. Subramanian, W. T. Tsai, R. Vishnuvajjala, R. Mojdehbakhsh, L. Elliott, “Testing Safety-Critical Systems-A Reuse-Oriented Approach” Proc. 9th Int. Conf. on SEKE, June, 1997, pp. 271-278.

[13] J. McGregor, “Testing a Software Product Line,” Testing Techn. in Softw. Eng. Springer Berlin Heidelberg, 2010.

[14] J. Bosch, Design and Use of Software Architecture: Adopting and Evolving a Product-Line Approach, Addison-Wesley, 2000.

[15] Condron. “A Domain Approach to Test Automation of Product Lines,” Int. Workshop on Softw. Product Line Testing. p 27, (2004).

[16] R. Tiwari. and N. Goel, “Reuse: Reducing Test Effort ACM,” SIGSOFT Softw. Eng. Notes, pp 1-11, March 2013

[17] M. Gupta and M. Prakash, “Possibility of Reuse in Software Testing,” 6th Annu. Int. Softw. Testing Conf. in India., 2006.

[18] D. Almog and T. Heart, "Developing the Basic Verification Action (BVA) Structure Towards Test Oracle Automation," IEEE 2010 Conf. on Computational Intell. and Soft. Eng. (CiSE), 2010, pp. 1-4.

[19] D. Almog and Y. Tesubery, “Reuse of Unit Test Artifacts – Allow Us to Dream,” Agile Rec. Issue 16 pp. 49 – 52, Nov. 2013.