

Environment Model-based System Testing of Real-Time Embedded Systems

Muhammad Zohaib Zafar Iqbal

Thesis submitted for the degree of Ph.D.

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
September 2012

© **Muhammad Zohaib Zafar Iqbal, 2012**

*Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1221*

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika publishing.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

Real-time embedded systems (RTES) are part of a vast majority of computing devices available today. They are widely used in critical domains where high system dependability is required. These systems typically work in environments comprising of large numbers of interacting components. The interactions with the environment are often bound by time constraints. Missing such time deadlines, or missing them too often for soft real-time systems, can lead to serious failures resulting in threats to human life or the environment. There is usually a great number and variety of stimuli from the RTES environment with differing patterns of arrival times. Testing all possible sequences of stimuli is not feasible and only a fully automated testing approach can scale up to the testing requirements of industrial RTES. In this thesis, we take a black-box approach for system testing of RTES based on environment models. Our main motivation is to provide a practical approach to the model-based testing (MBT) of RTES. To do so, we enable system testers, who are often not familiar with the system design but are knowledgeable of the application domain, to model the environment using well-supported modeling standards, to enable test automation. Once the environment models are developed they can support the automation of three tasks: the code generation of an environment simulator to enable testing on the development platform, the selection of test cases, and the evaluation of their expected results (oracles).

Given the above objectives, a first contribution of this thesis is a detailed environment modeling methodology that fits the practical needs for industrial adoption of a RTES system testing approach. The methodology is based on modeling standards (1) that are at an adequate level of abstraction, (2) that software engineers are familiar with, and (3) that are well supported by commercial or open source tools. The methodology uses the Unified Modeling Language (UML), the profile for Modeling and Analysis of Real-time Embedded Systems (MARTE), and the Object Constraint Language (OCL). We also provide extensions to UML and introduce a profile for modeling concepts that are specific to our context. The models capture only the details in the environment that are visible and relevant to the SUT, including the nominal behavior and failure behavior of environment components. The environment behavioral models also capture what we call ‘error states’ that should never be reached if the SUT is implemented correctly. The ‘error states’ act as oracles for the test cases. The environment modeling methodology is applied on two industrial case studies. The results show that the modeling notations selected suffice to

model the RTES environments for our test automation. The experiences learned by applying UML/MARTE in industrial contexts are also summarized in the form of a framework, which can help practitioners in bridging the gap between the modeling standards and industrial adoption.

A second contribution of this thesis is the definition of transformation rules for environment simulator generation. To convert environment models developed using UML state machines and class diagrams to their simulator code, we extend the well-known state pattern for our specific purpose and also resolve a number of UML semantic variation points. We evaluate the transformation rules by transforming models for five case studies, including two industrial case studies and use these models for testing. Our empirical evaluation based on the case studies shows that the developed rules are sufficient and that they are correct as far as fault detection is concerned. The automated simulator generation is expected to save a significant amount of effort during system testing.

The third contribution of this thesis is an efficient approach to solve constraints on the environment models written using OCL. For this purpose we define a set of heuristics for search algorithms and empirically evaluate their effectiveness on an industrial case study. These heuristics play an important role in test case generation from environment models. Results of the empirical study suggest that even for the most difficult constraints, with research prototypes and no parallel computations, we obtain test data within 2.96 seconds on average. This is a significant improvement compared to an existing OCL solver, which was not able to solve the same constraints even after several hours of execution.

The final contribution of the thesis is test case generation from environment models for black-box system testing of RTES. We conducted a number of experiments to investigate the effectiveness of testing algorithms, specifically, Genetic Algorithms (GA) and (1+1) Evolutionary Algorithm (EA), Adaptive Random Testing (ART), and Random Testing (RT) in our context. The goal of testing in our context is to reach an ‘error state’ of the environment with as few test case executions as possible. For search algorithms we provide and iteratively improve a fitness function for effective testing. The testing strategies are evaluated on an industrial case study and a number of artificial problems. On the industrial case study we were able to automatically find new, critical faults. Based on the results of our experiments, we propose a hybrid strategy, which combines the strengths of (1+1) EA and ART, to improve the overall performance of system testing that is obtained when using each single strategy in isolation. Results show that the hybrid strategy

fares better and, unlike individual algorithms, its performance is not drastically affected by the characteristics of the environment models (i.e., low variance in results).

Acknowledgements

First of all, I will like to thank my supervisors, Lionel Briand and Andrea Arcuri. Without their continuous guidance and support, this work would not have been possible. Lionel is an outstanding supervisor and has been a source of inspiration throughout my PhD. His passion for research has always motivated me. Andrea is an excellent supervisor and has always been a great help.

I will like to thank Simula Research Laboratory and Simula School of Research and Innovation for giving me the opportunity to work in an excellent environment. I would also like to thank people at our two industrial partners, WesternGeco and Tomra, who gave me the opportunity to work with industrial case studies that were fundamental in developing and evaluating my proposed testing approach.

I will also like to thank all my colleagues and friends at Simula, especially my office mate for three years, Amir Raza Yazdanshenas. Special thanks to my best friends Shaukat Ali, Tao Yue, Rajwinder Panesar-Walawega for all the fun time and discussions we had together. I would also like to thank Arnaud Gotlieb for all his help.

Last, but not the least, I will like to thank my family and friends for their continuous encouragement for my PhD. Special thanks to Prof. Zafar Malik who has always been a great mentor and a source of guidance and motivation. The acknowledgments will not be complete if I don't offer my gratitude to Prof. Jaffar-ur Rehman (late), who is still a great inspiration and had a key contribution to making me reach where I am today. My dear wife Amna, our son Zayaan, and my dearest Ammi and Abu Jan deserve my special thanks.

List of papers

The following papers are included in this thesis:

Paper 1. Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies

M.Z. Iqbal, A. Arcuri, L. Briand

In: Model Driven Engineering Languages and Systems (MODELS), vol. 6394, pp. 286-300. LNCS, Springer Berlin / Heidelberg, 2010

Paper 2. A Practical Approach to Environment Modeling and Simulation for Automated Testing of Real-Time Embedded Software

M.Z. Iqbal, A. Arcuri, L. Briand

Revision submitted to Software and Systems Modeling Journal (SoSyM), 2012.

Paper 3. Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing

A. Arcuri, M.Z. Iqbal, L. Briand

In: Testing Software and Systems (ICTSS), vol. 6435, pp. 95-110. LNCS, Springer Berlin / Heidelberg, 2010

Paper 4. Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software

M.Z. Iqbal, A. Arcuri, L. Briand

In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), pp. 199-209. ACM, 2012

Paper 5. Combining Adaptive Random Testing and Search-based Testing Strategies to Improve Environment Model-Based Testing of Real-Time Embedded Software

M.Z. Iqbal, A. Arcuri, and L. Briand

In: Search Based Software Engineering (SSBSE), vol. 7515, pp. 136-151. LNCS, Springer Berlin / Heidelberg, 2012

Paper 6. Experiences of Applying UML/MARTE on Three Industrial Projects

M.Z. Iqbal, S. Ali, T. Yue, and L. Briand

In: Model Driven Engineering Languages and Systems (MODELS), LNCS, Springer Berlin / Heidelberg, 2012 (*to appear*)

Paper 7. A Search-based OCL Constraint Solver for Model-based Test Data Generation

S. Ali, M.Z. Iqbal, A. Arcuri, and L. Briand.

In: Proceedings of International Conference on Quality Software (QSIC), pp. 41-50. IEEE, 2011

Paper 8. Generating Test Data from OCL Constraints with Search Techniques

S. Ali, M.Z. Iqbal, A. Arcuri, and L. Briand.

Submitted to IEEE Transaction on Software Engineering (TSE), 2012

The above papers are self-contained and therefore some information might be repeated across the papers. The papers may also use different abbreviations.

My contributions

For papers except Paper 3, Paper 7, and Paper 8, I was the main contributor. My supervisors (Andrea Arcuri and Lionel Briand) were involved throughout my PhD work. For Paper 1, I was the main contributor as I designed the modeling methodology and along with help of my supervisors applied it to two industrial case studies. For Paper 2, I was responsible for developing the transformation rules and the tool that generates the simulator along with writing of the paper. For Paper 4 and Paper 5, I was responsible for developing the improved testing strategies, designing and conducting the experiments, and analyzing the results. For Paper 6, I was the main contributor and along with Shaukat Ali and Tao Yue reported our industrial experiences of applying the modeling standards in three industrial applications. For Paper 3, I was involved throughout the various stages of the work, including paper writing, and experimentation. For Paper 7 and Paper 8, along with Shaukat Ali, I was the main contributor of the work and was involved in developing the heuristics, designing and conducting the experiments, and writing the paper.

During my PhD, I also worked on three other articles that are not included as part of my thesis. Paper 9 is related to the thesis as it evaluates test case representation for the testing we do and performance of ART, but it needs some additional work before it can be submitted for publication. Paper 10 (which got 2010 ACM Distinguished Paper Award) and Paper 11 theoretically assess random testing and are not included in the thesis as they do not fall in line with the focus of the thesis, which is environment model-based black-box system testing of real-time embedded systems.

Paper 9. Automated System Testing of Real-Time Embedded Systems Based on Environment Models

M.Z. Iqbal, A. Arcuri, and L. Briand

Simula Research Laboratory, Technical Report (2011)

Paper 10. Formal Analysis of the Effectiveness and Predictability of Random Testing

A. Arcuri, M.Z. Iqbal, and L. Briand

In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), pp. 219 – 230. ACM, 2010

Paper 11. Random Testing: Theoretical Results and Practical Implications

A. Arcuri, M.Z. Iqbal, and L. Briand

IEEE Transactions on Software Engineering (TSE), vol. 38(2), pp. 258-277. IEEE, 2012

Contents

Summary	1
1 Introduction.....	1
2 Background.....	5
2.1 Testing of Real-time Embedded Systems	5
2.2 Unified Modeling Language.....	6
2.3 Object Constraint Language (OCL).....	6
2.4 MARTE Profile	6
2.5 Search-based Testing.....	7
2.6 Adaptive Random Testing	8
3 Environment Modeling and Testing of Real-Time Embedded Systems	9
3.1 Environment Modeling.....	10
3.2 Environment Simulation.....	11
3.3 Environment Model-Based Testing.....	12
4 Research Methodology	14
4.1 Understanding Industrial Problems	14
4.2 Literature Survey	15
4.3 Developing Methodologies for Modeling, Simulator Generation & Testing	16
4.4 Empirical Studies.....	16
5 Summary of Results.....	17
5.1 Paper 1	17
5.2 Paper 2.....	18
5.3 Paper 3.....	20
5.4 Paper 4.....	21
5.5 Paper 5.....	24
5.6 Paper 6.....	25
5.7 Paper 7.....	26
5.8 Paper 8.....	26
6 Future Directions	28
7 Conclusion.....	29
8 References for the Summary.....	31

Paper 1. Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies.....

1. Introduction.....	33
2. Related Work.....	36
3. Environment Modeling - Methodology	37
3.1. Modeling Structural Details as Environment Domain Model	38
3.1.1. <i>Environment Components to be Included.</i>	39
3.1.2. <i>Relationships to be Included.</i>	39
3.1.3. <i>Properties to be Included.</i>	39
3.1.4. <i>Modeling the SUT.</i>	40
3.2. Modeling Behavioral Details with UML State Machines & MARTE.....	40
3.2.1. <i>Identifying Stateful Components.</i>	41
3.2.2. <i>States to be Included.</i>	41
3.2.3. <i>Modeling Users in the Environment.</i>	41
3.2.4. <i>Modeling Abstract Phenomena.</i>	42
3.2.5. <i>Modeling Transitions & Action Durations.</i>	42
3.2.6. <i>Modeling Non-Determinism.</i>	42
3.2.7. <i>Modeling Error & Failure States.</i>	44
3.3. Modeling the Constraints.....	45
3.4. Environment Modeling Profile.....	46
3.5. Simulation of Environment Models.....	46
4. Model-based Testing based on Environment Models	47
5. Case Studies.....	49
6. Conclusion	50
7. References.....	51

Paper 2. A Practical Approach to Environment Modeling and Simulation for Automated Testing of Real-Time Embedded Software	53
1. Introduction	53
2. Practical Aspects	57
3. Related Work	59
3.1. Modeling & Simulation for RTES Testing	59
3.2. Environment Modeling and Environment Model-based Testing	60
3.3. Code Generation from UML Classes and State Machines	61
3.4. Summary	63
4. Motivating Example	64
5. Environment Modeling Methodology	65
5.1. Environment Modeling Profile	66
5.2. Domain Modeling	67
5.2.1. <i>Environment Components to be Included</i>	69
5.2.2. <i>Relationships to be Included</i>	69
5.2.3. <i>Properties to be Included</i>	69
5.2.4. <i>Modeling the SUT</i>	71
5.3. Behavior Modeling	71
5.3.1. <i>Identifying Stateful Components</i>	72
5.3.2. <i>States to be Included</i>	72
5.3.3. <i>Modeling Users in the Environment</i>	73
5.3.4. <i>Modeling Events</i>	73
5.3.5. <i>Modeling Actions & Action Durations</i>	75
5.3.6. <i>Modeling Error & Failure states</i>	77
5.3.7. <i>Modeling Non-Determinism</i>	78
6. Simulator Generation	80
6.1. Simulation Framework	80
6.2. An Extended State Pattern for Environment Simulation	83
6.3. Transformation of the Domain Model	85
6.4. Transformation of Behavioral Models	87
6.4.1. <i>Event Handling</i>	87
6.4.2. <i>Handling Hierarchical State machines</i>	90
6.4.3. <i>Handling Non-Determinism</i>	92
6.4.4. <i>Handling Oracle Information</i>	93
6.4.5. <i>Handling Guards and Actions</i>	93
6.5. Various Design Decisions and Their Rationale	94
6.5.1. <i>Object Concurrency Model</i>	94
6.5.2. <i>Time Semantics</i>	94
6.5.3. <i>Execution Semantics and Order of Events in Queue</i>	95
6.5.4. <i>Default Entry & Handling Conflicting Triggers</i>	96
6.5.5. <i>Event not satisfying any Trigger</i>	96
6.5.6. <i>Event Evaluation Time</i>	96
6.5.7. <i>Signal Transmission</i>	97
6.6. Automation	98
6.7. Interaction with Test Framework	98
6.7.1. <i>Search Heuristics</i>	100
6.7.2. <i>Simulation Configuration</i>	102
6.7.3. <i>ACL Constraint Solver</i>	103
6.7.4. <i>Test Driver & JUnit Test Case</i>	104
7. Case Study	105
7.1. Case Study Design	105
7.2. Case study procedure	108
7.2.1. <i>Completeness of the Transformation Rules</i>	108
7.2.2. <i>Effect on Development Effort</i>	108
7.2.3. <i>Effectiveness in Test Automation</i>	108
7.2.4. <i>Correctness of Transformations</i>	109
7.2.5. <i>Completeness of the Modeling Methodology and Profile</i>	109
7.3. Results	109
7.3.1. <i>Completeness of the Transformation Rules</i>	110
7.3.2. <i>Effect on Development Effort</i>	110

7.3.3.	<i>Effectiveness in Test Automation</i>	112
7.3.4.	<i>Correctness of the Transformation Rules</i>	112
7.3.5.	<i>Completeness of the Modeling Methodology and Profile</i>	113
8.	Limitations.....	113
9.	Conclusion.....	114
10.	References.....	117

Paper 3. Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing..... 121

1.	Introduction.....	121
2.	Related Work.....	124
3.	Environment Modeling and Simulation.....	125
4.	Automated Testing.....	126
4.1.	Test Case Representation.....	126
4.2.	Testing Strategies.....	128
5.	Empirical Study.....	131
5.1.	Case Study.....	131
5.2.	Experiments.....	132
5.3.	Discussion.....	134
5.4.	Practical Guidelines.....	136
6.	Threats to validity.....	137
7.	Conclusion.....	138
8.	References.....	139

Paper 4. Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software..... 142

1.	Introduction.....	143
2.	Background.....	144
3.	Related Work.....	145
4.	Environment Modeling and Model-based Testing.....	147
4.1	Environment Modeling & Simulation.....	147
4.2	Environment Model-Based Testing.....	148
5.	Improved Fitness Function.....	151
5.1	Improved Time Distance (ITD).....	152
5.2	Time in Risky State (TIR).....	153
5.3	Risky State Count (RSC).....	153
5.4	Increase in Coverage (COV).....	154
5.5	Combination of heuristics.....	154
6.	Empirical Study.....	155
6.1	Case Study.....	155
6.2	Experiments.....	157
6.3	Results and Discussion.....	159
6.4	Threats to validity.....	166
7.	Conclusion.....	167
8.	References.....	168

Paper 5. Combining Adaptive Random Testing and Search-based Testing Strategies to Improve Environment Model-Based Testing of Real-Time Embedded Software..... 170

1.	Introduction.....	170
2.	Related Work.....	173
3.	Environment Model-based Testing.....	174
3.1.	Environment Modeling & Simulation.....	174
3.2.	Testing RTEs based on Environment Models.....	175
4.	Hybrid Strategy by Combining Adaptive Random and Search-based Testing.....	179
5.	Empirical Study.....	181
5.1.	Case Study.....	181
5.2.	Experiment.....	182
5.3.	Results & Discussion.....	185
5.4.	Threats to Validity.....	188
6.	Conclusion.....	188

7. References	189
Paper 6. Experiences of Applying UML/MARTE on Three Industrial Projects.....	191
1. Introduction	191
2. Background.....	193
3. Industrial Applications of UML/MARTE	194
3.1. Architectural Modeling and Configuration with UML/MARTE.....	194
3.1.1. Case Study Description.....	194
3.1.2. Problem Description.....	194
3.1.3. Modeling Solution.....	195
3.1.4. Modeling Tool.....	195
3.1.5. Key results	196
3.2. Model-based Robustness Testing with UML/MARTE	196
3.2.1. Case Study Description.....	196
3.2.2. Problem Description.....	197
3.2.3. Modeling Solution.....	197
3.2.4. Modeling Tool.....	198
3.2.5. Key Results.....	198
3.3. Testing RTES using UML/MARTE environment models	199
3.3.1. Case Study Description.....	199
3.3.2. Problem Description.....	199
3.3.3. Modeling Solution.....	200
3.3.4. Modeling Tool.....	201
3.3.5. Key Results.....	201
4. Framework for Applying UML/MARTE in Industry.....	202
4.1. Perform Domain Analysis (A1).....	202
4.2. Define a Modeling Methodology (A2)	204
4.2.1. Identify Notations (A2.1).....	204
4.2.2. Extend Notations (A2.2).....	205
4.2.3. Tool Selection (A2.3).....	207
4.2.4. Define Guidelines (A2.4).....	208
4.3. Application of Methodology.....	208
4.4. Summary and Discussion	209
5. Conclusion.....	210
6. References	210
Paper 7. A Search-based OCL Constraint Solver for Model-based Test Data Generation.....	213
1. Introduction	213
2. Background.....	215
3. Related Work.....	215
3.1. OCL-based Constraint Solvers.....	216
3.2. Search-based Heuristics for Model Based Testing.....	217
4. Definition of the Fitness Function for OCL.....	217
4.1. Primitive types	219
4.2. Collection-Related Types.....	220
4.3. Equality of collections (=).....	221
4.4. Operations checking existence of one or more objects in a collection	221
4.5. Branch distance for iterators.....	221
5. Case Study: Robustness Testing Of Video Conference System	222
5.1. Empirical Evaluation.....	224
5.1.1. Experiment Design.....	224
5.1.2. Experiment Execution	225
5.1.3. Results and Analysis	225
5.2. Comparison with UMLtoCSP	227
6. Tool Support.....	227
7. Threats to Validity.....	228
8. Conclusion.....	229
9. References	230

Paper 8. Generating Test Data from OCL Constraints with Search Techniques	233
1. Introduction.....	233
2. Background.....	235
3. Related Work.....	236
3.1 Comparison with OCL Constraints Evaluation	236
3.2 OCL-based Constraint Solvers	237
3.3 Search-based Heuristics for Model Based Testing	240
4. Definition of the Fitness Function for OCL	241
4.1 Primitive types.....	242
4.2 Collection-Related Types	244
4.2.1 Equality of collections (=).....	245
4.2.2 Operations checking existence of one or more objects in a collection.....	248
4.2.3 Branch distance for iterators	249
4.3 Tuples in OCL.....	256
4.4 Special Cases	256
4.4.1 Enumerations	256
4.4.2 <i>oclInState</i>	257
4.4.3 <i>oclIsTypeOf()</i> , <i>oclIsKindOf()</i> , and <i>oclIsNew()</i>	259
4.4.4 <i>User-defined Operations</i>	259
5. Case study: Robustness Testing of Video Conference System	260
5.1 Empirical Evaluation	261
5.1.1 <i>Experiment Design</i>	262
5.1.2 <i>Experiment Execution</i>	262
5.1.3 <i>Results and Analysis</i>	267
5.2 Comparison with UMLtoCSP.....	269
6. Empirical Evaluation of Optimization Defined as Fitness Functions	271
6.1 Experiment Design	271
6.2 Experiment Execution	273
6.3 Results and Analysis.....	273
7. Overall Discussion.....	275
8. Tool Support	277
9. Threats to Validity	277
10. Conclusion	279
11. References.....	280

Summary

1 Introduction

Real-time embedded systems (RTES) are widely used in many different domains, from integrated control systems to consumer electronics. Already 98% of computing devices are embedded in nature and it is estimated that, by the year 2020, there will be over 40 billion embedded computing devices worldwide [1]. These systems typically work in environments comprising of large number of physical components (e.g., sensors and actuators) and possibly other RTES systems (e.g., in systems of systems). The interactions with the environment are usually bounded by timing constraints. For example, if a gate controller RTES on a railroad intersection is informed by a sensor that a train is approaching, then the RTES should command the gate to close before the train reaches it. Missing such time deadlines, or missing them too often for soft real-time systems, can lead to serious failures leading to threats to human life or the environment. There is usually a great number and variety of stimuli from the RTES environment with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. Testing all possible sequences of stimuli is not feasible. Hence, systematic automated testing strategies that have high fault revealing power are essential for effective testing of industry scale RTES.

Because RTES are developed for diverse domains presenting different characteristics (e.g., different timing, safety, security requirements), different testing approaches are required to handle the significant variation across domains [2]. Our main target RTES in this thesis are soft-real time systems with time deadlines in the order of hundreds of milliseconds, with an acceptable jitter of a few milliseconds in response time. Our testing

approach (black-box system level testing) not only encompasses functional correctness of the system under test (SUT), but also enable to focus testing on particularly critical aspects of the RTES, e.g., potentially hazardous situations.

The work discussed in this thesis was motivated by the problems faced and practices followed by two industrial organizations that we worked with, namely WesternGeco AS, Norway [3] and Tomra AS, Norway [4]. These two organizations were developing RTES for two different domains; WesternGeco was developing a seismic acquisition system and Tomra was developing automated recycle machines. Both the RTES were developed to run in an environment that enforces time deadlines in the order of hundreds of milliseconds with an acceptable jitter of a few milliseconds in response time. In one of the organizations, testing the SUT on the development platform with a simulated environment was considered to be mandatory before deploying the software on the operational hardware. To achieve this, software engineers were writing application specific simulators directly in Java. Test cases for system level testing were written by hand by the software test engineers and were executed on the SUT with the environment simulator. The research presented in this paper was strongly driven by our investigation of the practical needs of our industry partners which, based on our experience, are shared by many others in numerous industry sectors.

Typically, large scale testing of RTES software in real environments and on actual deployment platforms is not a viable option. It would be expensive, the consequences of failures might be catastrophic (e.g., in safety critical systems), and the number of variations in the environment that can be exercised within a reasonable time frame are small. Moreover, some of the environment components might not be available at the time of testing, since hardware and software components are typically developed concurrently. To test RTES software in this kind of situations, a common strategy is to develop a simulator for these environment components. A simulator enables the execution of the RTES on the development platform, without requiring actual interactions with its environment. In our context, a test case is a sequence of stimuli, generated by the environment or its simulator, that are sent to the RTES. If a user interacts with the RTES, then the user would be considered as part of the environment as well.

Testing all possible sequences of environment stimuli and state changes is not feasible. In practice, a single test case of an industrial RTES could last several seconds or even minutes, executing hundreds of thousands of lines of code, generating hundreds of threads and processes running concurrently, communicating through TCP sockets and operating

system signals, and accessing the file system for I/O operations. Hence, systematic testing strategies that have high fault revealing power must be devised. The complexity of modern RTES makes the use of systematic testing techniques, whether based on the coverage of code or models, difficult to apply without generating far too many test cases. Alternatively, manually selecting and writing appropriate test cases based on human expertise for such complex systems would be far too challenging and time consuming. If any part of the specification of the RTES changes during its development, a very common occurrence in practice, then many test cases might become obsolete and their expected output would potentially need to be recalculated manually. The use of an automated oracle is hence another essential requirement when dealing with complex industrial RTES.

In this thesis, we present a practical approach for automated system testing of RTES based on its environment models. The main contributions of this thesis are as follows:

- We propose a methodology for modeling environments of RTES for automated system testing by using international modeling standards: the Unified Modeling Language (UML) [5], the Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile [6] and our proposed profile for environment modeling (discussed in Paper 1). The proposed methodology is applied on two industrial case studies. Based on our experiences in industrial applications of our methodology, we derive a framework to help modelers for future industrial applications of UML/MARTE. The framework provides a set of detailed guidelines on how to apply these standards in industrial contexts and will help reduce the gap that is to be expected between such modeling standards and industrial needs
- We present extensions to the *state pattern* [7] specifically aimed at enabling environment simulation for system testing and define rules for transforming environment models to Java code (the simulator). The rules are empirically evaluated for two industrial case studies and three artificial problems
- A testing approach that uses the environment models to automatically generate test cases and test oracles for RTES system testing. We tailored ART and defined specific fitness functions for search-based testing (SBT). For applying SBT, a fundamental requirement was to solve OCL constraints in the UML models. To fulfill this need the thesis proposes heuristics for the application of SBT to solve these constraints. We empirically evaluated these techniques on one industrial case study and a number of artificial problems. The results of these evaluations lead us to propose a hybrid strategy that provides the benefits of both ART and SBT. The results of our experiments to

evaluate the fault detection effectiveness of this hybrid strategy suggest that it is a practical strategy to apply, since unlike other strategies, variations in environment properties do not have a drastic impact on its performance. This makes it a predictable test strategy.

- Finally, we report our experience of applying UML/MARTE for model-based testing in industrial contexts and based on such experiences, we propose a framework to guide future practitioners on applying UML/MARTE in industry.

This thesis has two parts:

Summary: This part provides an overall summary of the thesis and is organized as follows: Section 2 provides the necessary background information required for the thesis. Section 3 summarizes the contributions of the thesis, whereas Section 4 discusses the research methodology that was followed. Section 5 highlights the results of the research papers that are submitted as part of the thesis. Section 6 provides future research directions and finally Section 7 concludes the thesis.

Papers: This part provides the published or submitted research papers that are included in this thesis.

2 Background

This section provides the background of the work reported in this thesis.

2.1 Testing of Real-time Embedded Systems

Depending on the goals, RTES testing can be performed at different levels. At the early stages of the development process for RTES, a typical approach is to model and simulate the SUT, its hardware and its environment. The aim is to ensure that the model of the SUT complies with the requirement specifications and does not violate the environment and hardware assumptions. This approach is sometimes also referred to as “model-in-the-loop” simulation or testing [2, 8, 9]. Another level of testing is when the actual executable software is deployed on the real hardware platform (e.g., electronic control unit) and their combination is tested with a simulated environment (e.g., with the simulation of a plant model [2]). This approach is generally called “hardware-in-the-loop” testing [10, 11]. Typically, a prototype of the hardware platform is used at this stage. A variation to hardware-in-the-loop testing is the case where only the actual processor is used during testing and the rest of the hardware and environment are simulated. This variation is widely referred to as “processor-in-the-loop” testing [12].

Before the hardware or the processor is available, the embedded software can also be tested on the development platform (e.g., Linux or Windows-based machine) with a simulated environment and hardware platform. This is typically done to ensure that the developed software works according to the environment assumptions and in hazardous situations. This is mostly referred to as “software-in-the-loop” [2, 8]. Existing modeling and simulation languages and their corresponding testing techniques have been developed and are widely used for the first three types of testing. In these cases the environment simulation needs to interact with the actual hardware or its simulation. In such cases, precise simulation of both discrete and continuous phenomena is required and is typically based on mathematical models.

The approach presented in this thesis can be labeled as a slight variation of the typical software-in-the-loop testing as we only model and simulate the environment to test the SUT. We use an adapter for the hardware platform that forwards the signals from the SUT to the simulated environment.

2.2 Unified Modeling Language

Unified Modeling Language (UML) [5] is an international standard for modeling different aspects of software systems. With a total of 13 diagrams in UML 2.x, the language enables the modelers to represent software systems at various levels of abstraction. For modeling the static structure of such systems, it provides class diagram, object diagram, package diagram, component diagram, composite structure diagram, deployment diagram, and profile diagram. For modeling the behavior UML provides with use case diagram, activity diagram, state machine diagram, sequence diagram, communication diagram, interaction overview diagram and timing diagram. Depending on the system being model and the purpose of modeling, typically a methodology is defined which identifies the subset of UML to be used. UML also provides a built-in mechanism to provide lightweight extensions that do not conflict with its original semantics by developing UML profiles.

2.3 Object Constraint Language (OCL)

OCL [13] is an international standard language for writing constraints on UML models. It is a textual language and is based on first order logic and set theory, but is more expressive as its syntax is closer to higher level programming languages. Since, it is a specification language, the expression written in OCL do not have any side effects. Depending on the goals, constraints can be written for different elements of UML models, ranging, for example, from class invariants to guards on state machines. A subset of this language can also be used to define constraints on meta-models, which for example is used to define UML meta-model. The language also provides a standard library that defines a number of operations on various OCL types, including collections, that are helpfull when writing constraints.

Constraints written on UML models, as for example, the constraints written as part of guard conditions on state transitions in state machines, play an important role during model-based testing. As an example, consider a testing scenario where transition coverage based on a UML state machine is required. If any of the transitions in the state machine is guarded (where the guard is written in OCL), then to achieve the required coverage, the guard needs to be satisfied in order to trigger the transition.

2.4 MARTE Profile

The UML profile for Modeling and Analysis of Real-time Embedded Systems (MARTE) [6] was defined to provide a number of concepts that modelers can use to

express relevant properties of RTES, for example related to performance and schedulability. MARTE is meant to replace the previously defined UML profile for Schedulability, Performance, and Time specification (SPT) [14].

At the highest level, MARTE contains three packages. The core package is MARTE Foundations that contains the sub-packages for modeling non-functional properties (NFP package), time properties (Time package), generic resource modeling of an execution platform for RTES (GRM package), and resource allocation (Alloc package). The MARTE Foundations package contains the core elements that are reused by the other two packages of the profile: MARTE design model and RealTime&Embedded Analysing (RTEA). The MARTE design model package contains various sub-packages required for modeling the design of RTES. This includes the packages to support modeling of component-based RTES with the Generic Component Model package (GCM), high-level features for RTES with the High-Level Application Modeling package (HLAM), and for detailed modeling of software and hardware resources with the Detailed Resource Modeling package (DRM). The RTEA package contains further concepts related primarily to modeling for analysis. This includes the Generic Quantitative Analysis Modeling package (GQAM) which provides generic concepts for resource modeling. These concepts are further specialized by the Schedulability Analysis Modeling (SAM) package for modeling properties useful for Schedulability and the Performance Analysis Modeling package (PAM) for modeling properties useful for performance analysis.

2.5 Search-based Testing

Several software engineering problems can be reformulated as a search problem, such as test data generation [15]. An exhaustive evaluation of the entire search space (i.e., the domain of all possible combinations of problem variables) is usually not feasible. There is a need for techniques that are able to produce “good” solutions in reasonable time by evaluating only a tiny fraction of the search space. Search algorithms can be used to address this type of problem. Several successful results by using search algorithms are reported in the literature for many types of software engineering problems [16].

To use a search algorithm, typically a fitness function needs to be defined. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. The fitness function will be used to guide

the search algorithms toward fitter solutions. Eventually, given enough time, a search algorithm will find a satisfactory solution.

There are several types of search algorithms. Genetic Algorithms (GA) are the most well-known [16], and they are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through crossover and mutation operators. (1+1) Evolutionary Algorithm (EA) is simpler than GA, in which only a single individual is evolved with mutation. To verify that search algorithms are actually necessary because they address a difficult problem, it is a common practice to use Random Search (or Random Testing (RT) for testing problems) as a comparison baseline [16].

2.6 Adaptive Random Testing

Adaptive Random Testing (ART) [17] has been proposed as an extension of RT. The underlying idea of ART is that diversity among test cases should be rewarded, because failing test cases tend to be clustered in contiguous regions of the input domain. ART can be automated if one can define a meaningful similarity function for test cases.

3 Environment Modeling and Testing of Real-Time Embedded Systems

The main motivation of the thesis is to provide a practical approach for automated black-box system testing of RTES based on their environments. Fig. 1 shows a high level view of the framework for RTES system testing. The major input required by the software engineer is the environment models. These models are then translated using the simulator generator to a Java simulator. The software engineer also writes a minimal driver that configures the *Test Framework*. Environment models comprise of a domain model and a number of behavioral models. The domain model represents the overall structure of the environment, shown as a UML class diagram. The behavioral models represent the behavior of environment components using UML 2.x state machines. The *Simulator Generator* component shown in the figure generates a set of Java files implementing a *Simulator* for the environment. A set of classes labeled as *External Action Code* contain the code written by the tester containing complex actions and communication related code between the SUT and its environment (e.g., through UDP/TCP, as it was the case in both our industrial case studies). An *OCL Constraint Solver* is used during simulator generation to resolve any constraints on the environment models in order to generate values for environment components' attributes. Later the constraint solver is embedded within the generated simulator and during simulation it calculates how far a test case is from satisfying a guard on a transition (i.e., the branch distance used to guide the search algorithms). The *Test Framework* is responsible for generating various test cases and starting up the RTES under test and the environment for each test case. The framework

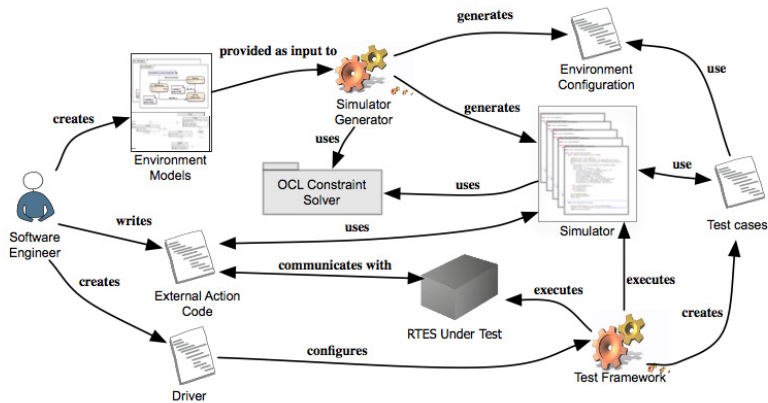


Fig. 1. Framework for Environment Model-based Testing of RTES

allows test case generation using three testing strategies (RT (random testing), ART (adaptive random testing), SBT (search-based testing), and a hybrid strategy combining SBT and ART).

The thesis can be divided into three related parts: (i) methodology for environment modeling; (ii) simulator generation from environment models; (iii) strategies for testing based on environment models including test data generation from OCL constraint. In the following sections, we give a brief overview of these three parts.

3.1 Environment Modeling

The first step is to model the characteristics and behavior of the environment. Environment models describe both relevant structural and behavioral characteristics of the environment. Given an appropriate level of detail, defined by our methodology, the models enable the automatic generation of the environment simulator. These models can also be used to generate automated test oracles, which are typically modeled as “error states” that should never be reached by the environment during the execution of a test case. From a practical standpoint, using the same model as the source for generating simulators and test cases is very important. Moreover, the models can further be used to automatically select test cases and sophisticated heuristics are used to automatically do so from the models without any intervention of the tester. To summarize, the only required artifacts to be developed by testers is the environment model and the rest of the process is expected to be fully automated. Incidentally, by using this automated Model-Based Testing (MBT) technology, one of our industrial partners was able to find new critical faults in their already tested RTES.

To support environment modeling in a practical fashion, we have selected standard and widely accepted notation for modeling software systems, the UML and its standard extensions. We use the MARTE [6] extensions for modeling real-time features and OCL for specifying constraints. We have also provided lightweight extensions to UML as a profile, in order to ease its use in our context. The corresponding profile diagram is shown in Fig. 2. Modeling the environment of industrial RTES using a combination of UML, MARTE, and OCL has not been addressed in the literature. By using the proposed methodology, software testers (who are primarily software engineers) can model the environment with a notation that they are familiar with, using commercial or open source tools, and at a level of precision required to support automated MBT. The importance of

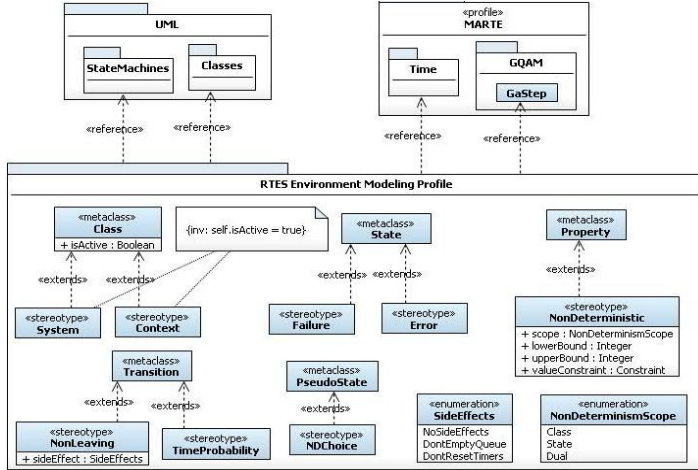


Fig. 2. RTEs Environment Modeling Profile

relying on standards for modeling was confirmed on the two industrial case studies across entirely different domains.

While modeling our industrial cases, we abstracted the functional details of the environment components to such an extent that only the details visible to the SUT were included. An environment of a RTEs typically features a number of non-deterministic events (e.g., breakdown of a sensor), which must be modeled. Such events are not common when modeling the internal behavior of a system.

In the kind of testing this thesis addresses, the focus is on the interactions of the RTEs with the components in its environment, i.e., what are the possible inputs/outputs to/from the RTEs from/to these components at any given point in time? How does the RTEs behave in abnormal situations, such as a hardware failure in any of the environment components? A test case for a RTEs would typically consist of a sequence of actions from the user(s), signals from/to sensors/actuators, and possibly hardware component breakdowns. This would correspond, in our context, to non-deterministic events that can happen during the environment simulations.

3.2 Environment Simulation

Although code generation from models has been widely studied, the context of black-box RTEs system testing poses specific challenges and problems that are not fully discussed and addressed in the literature. For this purpose we provide extensions to the original state pattern [7] specifically aimed at enabling environment simulation for system testing and define rules for transforming environment models to Java code (the simulator).

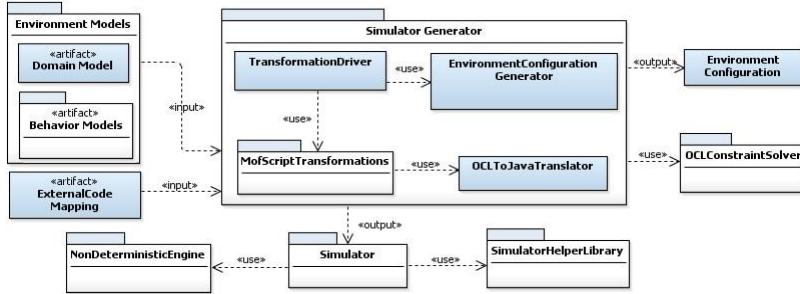


Fig. 3 Architecture Diagram of Simulation Framework

The original state pattern did not address a number of important features of UML state machines, such as, concurrency, parallel regions, composite states, time events, change events, and actions (on transition & within state). A number of extensions for the pattern have been discussed over time to handle missing features (e.g., [18, 19]). But overall, as discussed in Paper 2, none of the extensions completely meet the needs for RTES environment simulation to support system testing. We have adopted the extensions proposed in the literature wherever they were adequate for our needs. We also resolved a number of UML 2.x semantic variation points related to class diagrams and state machines for code generation. The model-to-text transformations for generating Java simulators from environment models are written using MofScript [20]. Fig. 3 shows an architecture diagram for the simulation framework. The domain model and behavioral model are inputs and a simulator corresponding to the environment models is one of the outputs of the framework. Further explanation of the framework is provided in Paper 2.

3.3 Environment Model-Based Testing

For model-based test case generation, we tailored the principles of Adaptive Random Testing (ART) [17] and Search-based Testing (SBT) [21] (specifically Genetic Algorithms and (1+1) Evolutionary Algorithm) to our specific problem and context. For our empirical evaluation, we also used Random Testing (RT) as a comparison baseline. One main advantage of ART and SBT is that they can be tailored to whatever time and resources are available for testing: when resources are expended and time is up, we can simply stop their application without any side effect. Furthermore, ART and SBT attempt, through different heuristics, to maximize the chances to trigger a failure within time constraints.

In our context, a test case is a sequence of stimuli generated by the environment that is sent to the RTES that can be taken during the simulation. If a user interacts with the RTES, then she would be considered part of the environment as well. A test case can also include

state changes in the environment that can affect the RTES behavior. For example, with a certain probability, some hardware components might break, and that affects the expected and actual behavior of the RTES. A test case can contain information regarding when and in which order to trigger such changes. So, at a higher level, a test case in our context can be considered as a setting specifying the occurrence of all these environment events in the simulator. Explicit “error” states in the models represent states that should never be reached if the RTES is correct. If any of these error states is reached, then it implies a faulty RTES. Error states act as the oracle of the test cases, i.e., a test case is successful in triggering a failure in the RTES if an error state of the environment is reached during testing.

A fundamental part of the fitness functions devised for SBT is the branch distance. In our context, the branch distance heuristically evaluates how close the values of a test case are to satisfy a guard on a transition on environment behavior models. Since the guards are written in OCL, we developed an OCL constraint solver for this purpose. Paper 7 and Paper 8 discusses the constraint solver in detail.

Our focus is to devise a practical approach in a system testing context. For this purpose, we evaluate the proposed modeling methodology and simulation generation on two industrial case studies. The proposed testing methodology is evaluated on an industrial case study and a set of artificial problems inspired by two industrial case studies.

4 Research Methodology

This thesis reports on industry-driven research aimed at finding applicable solutions to real, carefully defined problems. Defining such problems, their solutions and evaluations were, in our context, performed in collaboration with Tomra [4] and WesternGeco [3]. The research methodology followed for this thesis included understanding industrial problems in context, assess existing related work in terms of addressing the defined problems, developing specific modeling and testing methodologies, developing a tool for simulation and testing, conducting empirical studies for evaluating the methodologies and the tool, and iteratively improving the methodology and the tool based on the results of these empirical studies.

4.1 Understanding Industrial Problems

The thesis started by understanding the testing problems faced by our industry partners. These two partners were developing RTES for two different domains; WesternGeco was developing a seismic acquisition system and Tomra was developing automated recycle machines. Both the RTES were developed to run in an environment that enforces time deadlines in the order of hundreds of milliseconds with an acceptable jitter of a few milliseconds in response time. In one of the organizations, testing a SUT as a black-box on the development platform with a simulated environment is considered to be mandatory before deploying the software on the operational hardware. This is preferably carried out by independent testers who are application domain experts, but have little or no knowledge of SUT design and implementation. To achieve this, software engineers were writing application specific simulators directly in Java. Test cases for system level testing were written by hand by the software test engineers and were executed on the SUT with the environment simulator. In practice, a single test case for the type of testing done by our industry partners lasts several seconds, executing thousands of lines of code, generating hundreds of threads/processes running concurrently, communicating through TCP sockets and/or OS signals, and accessing the file system for I/O operations.

Testing all possible sequences of environment stimuli/state changes is not feasible. Manually selecting and writing appropriate test cases based on human expertise for such complex systems was very challenging and time consuming. If any part of the specification of the RTES changed during its development, a very common occurrence in practice, then

many test cases became obsolete and their expected output was to be recalculated manually.

Manually writing an environment simulator using a programming language (e.g., Java or C) for system testing also posed a number of issues, the main one being that software engineers had to develop such simulator at a low-level of abstraction while simultaneously focusing on the logic of the simulator, complex programming constructs (e.g., multiple threads, handling timers), and the handling of test case configurations (when the simulator is used for testing). Making this problem even more acute, over the course of the RTES development, these simulators frequently changed due to changes in the specifications of the hardware components.

To solve the identified industrial problem, there was a need to devise a systematic testing methodology that has high fault revealing power. The target systems of the methodology are RTES having complex environments and soft-real time constraints in the order of hundreds of milliseconds pertaining to the response time of the SUT and operations of the environment. The developed methodology should be adaptable and scalable to the specific complexity of a RTES and available testing resources. To enable complete automation of the testing methodology, an automated oracle is also required. The methodology should not only generate meaningful test cases based on RTES environments, but should also generate automated simulators for the environments, preferably from same set of specifications/models. The methodology should also be easily transferable to software engineers working at industry partners in a way that minimum specialized training is required.

The industrial case studies that we worked on are discussed throughout the papers (esp. Paper 2) and the steps followed along with our experiences to understand the problems and devise a solution are reported in Paper 6.

4.2 Literature Survey

Next step after understanding and precisely defining the industrial problem to target was to survey the literature for works (partially) matching our problem. We did not find any existing work that was entirely adequate for the needs of our industry partners. The environment modeling approaches reported in the literature were not based on modeling standards and not focusing on the type of testing we automate, i.e., automated black-box system testing of RTES. A number of approaches reported in the literature discuss code generation from models, but none of the approaches fit the requirement of generating

simulators for supporting the system testing of RTES. The results of the survey are discussed in Paper 2 and Paper 4.

4.3 Developing Methodologies for Modeling, Simulator Generation & Testing

After understanding the industrial problem and conducting the literature survey, the next step was to develop a modeling methodology for supporting system test automation in our context. For this we looked at the needs of the industry partners and tried to adopt standard modeling notations to the maximum extent possible. The step was iterative and involved feedback from the industry partners and also accommodating the new requirements that arose after developing the simulation and testing strategies. For simulator generation, we extended the well-known state pattern [7] according to our needs. For testing, we followed a step-wise strategy to obtain the best strategy for the type of testing that we perform. This was based on the results of extensive empirical studies that we carried out.

4.4 Empirical Studies

A fundamental part of the thesis was to carry out empirical studies to evaluate and later improve the methodologies for modeling, simulation, and testing. For modeling and simulation, we applied our methodology and simulator generation rules on two industrial case studies. For empirically evaluating our testing strategies, we developed thirteen artificial problems based on these two case studies and a case study discussed in the literature. We carried out a number of experiments on these artificial problems and the industrial case study of WesternGeco, which are discussed in Paper 3, Paper 4, and Paper 5. For evaluating our OCL constraint solver in isolation, we conducted a set of experiments on another industrial case study by Cisco Systems [22], which is discussed in Paper 7 and Paper 8.

5 Summary of Results

In this section, a summary of the key results of the papers submitted as part of this thesis are presented.

5.1 Paper 1

In this paper, we proposed methodology for modeling the environment of a RTES in order to enable black-box, system test automation. For practical reasons and to facilitate its adoption, the methodology is based on standards: UML, MARTE profile, and OCL for modeling the structure, behavior, and constraints of the environment. We, and this is part of our methodology, made a conscious effort to minimize the notation subset used from these standards. The paper also discusses the profile that we proposed for modeling the environment. The profile provides extensions to UML to model concepts specific to our approach, including non-determinism – an important characteristic of RTES environments.

The methodology provides in depth guidelines on how to model the environment structure and behavioral details. The structural details of the environment are captured using domain model. A domain model captures the structural details of the RTES environment, such as the environment components, their relationships, and their characteristics.

The behavior of the environment components is captured by state machines. To minimize modeling effort, the methodology aims at capturing only the details in the environment which are visible and relevant to the SUT. This not only includes the nominal functional behavior of the environment components (e.g., booting of a component) but also includes their robustness (failure) behavior (e.g., break down of a sensor). The latter are modeled as failure states in the environment models. The environment behavioral models also capture what we call error states. These are the states of the environment that should never be reached if the SUT is implemented correctly (e.g., no incorrect or untimely message from the SUT to the environment component). Error states act as oracles for the test cases. For example, recall the example of a system under test that controls a physical gate on a railroad intersection. The gate should always be down whenever a train is reaching the intersection and should be raised in other situations. The various trains approaching the intersection and the gate will together compose the environment of the SUT. The domain model will comprise of a train component, a gate component, and the SUT. A state machine each for the train and gate components will specify their behavior. A

possible failure state can for example be when the physical gate is stuck in a position (in which case the trains should be stopped before reaching the intersection) and a possible error state can be the situation when a train arrives at the gate while it is still open.

An important feature of these environment models is that they capture the non-determinism of the environment, which is a common characteristic for RTES environments (for example, the time it takes to change a gate position can have a variation of few seconds). Non-determinism may include, for example, different occurrence rates and patterns of signals, failures of components, or user commands. The environment modeling profile provides special constructs to model non-deterministic behavior of the environment. Each environment component can have a number of non-deterministic choices whose exact values are selected at the time of testing. Java is used as an action language and OCL is used to specify constraints and guards.

We modeled the environments of two industrial RTES in order to investigate whether our methodology and the notation subsets selected were sufficient to fully address the need for automated system testing. Results suggested that the methodology was sufficient to model the details at a level of abstraction that could be used to generate environment simulators, meaningful test cases, and obtain test oracles. Notations provided by UML, MARTE, and our proposed profile were sufficient to model the details required by these case studies (belonging to different domains).

5.2 Paper 2

This paper is a journal extension of Paper 1 with the following differences:

1. The environment modeling profile has been extended based on the needs of more sophisticated testing strategies
2. A discussion on how the various UML semantic variation points (related to the models being used) are resolved is added
3. Rules for generating the executable simulator from the environment models and its integration with the test framework are discussed in detail
4. The empirical evaluation has been improved in the following ways:
 - a. Three new artificial problems inspired from industrial case studies have been added to further evaluate the modeling methodology and simulator generation
 - b. The models of industrial cases have been modified according to the extended profile. Only that subset of industrial cases that is later used for testing is discussed.

c. The evaluation of transformation rules for simulator generation has been added

Apart from discussing the environment modeling methodology, the paper also discusses the transformation rules to convert environment models into a Java-based simulator. The rules are based on an extension of the state pattern [7], which is a well-known way of implementing state machines. The transformations proposed in the paper are defined to address the specific requirements for environment simulation and RTES system testing. The rules discussed include rules for transformation of association, attributes, and non-determinism modeled in the domain model and rules for transforming various state machine elements, including various events, hierarchical state machines, and non-determinism, to their corresponding Java code. A number of design decisions including the ones that were taken to resolve open semantic variations points of UML are also discussed in the paper. We followed the Active object model [5] to handle object concurrency. This is because they operate independently in the RTES environment and can communicate asynchronously with each other and the SUT. These objects have their own thread of execution and receive asynchronous messages that are handled using an event queue.

The following research questions are addressed in this paper:

RQ1. Are the transformation rules sufficient to convert environment models of different complexity levels, and belonging to various domains, to simulator code?

According to results of generating simulators from environment models for five different cases, including two industrial case studies, the transformation rules are complete. These test models along with the three artificial problems and two industrial cases covered all the modeling elements defined in the methodology. The MOFScript transformations developed were able to generate Java code for all of the UML/MARTE/OCL model constructs used in the case study artifacts and the test models.

RQ2. Is the automated generation of simulators likely to significantly reduce development effort?

Based on our experiences of working with two industrial case studies, we expect that the automated generation of the simulator code can save significant effort to the developers. Though there is a considerable effort involved in developing environment models, given the amount and complexity of the source code generated, it is expected to be less than the effort required for manually developing and maintaining environment simulator code with concurrency and complex synchronization issues.

RQ3. How effective is the generated simulator in enabling the detection of failures in RTES system testing?

With the generator simulators, the testing framework was able to trigger system failures corresponding to all the seeded faults in the problems. For an industrial case study, the testing was able to find a previously undetected critical fault in the RTES. Taken together, the results of these experiments increased our confidence that the generated simulators are effective in detecting faults in the SUT when used in combination with various test automation strategies.

RQ4. Are the transformations implemented correctly?

The results of experiments conducted, manual inspection, and initial testing of the generated code suggest that it was generated according to the environment models following the extended state pattern.

RQ5. Are the proposed methodology and profile sufficient for modeling environments of RTES for the type of testing we are interested in?

For all the five cases, we were able to model the RTES environments with the subset of UML and MARTE that we identified and the lightweight extensions that we proposed. The models were sufficient to generate simulators that could be used to support large-scale test automation. The results of testing the five RTES show that the notations are sufficient for the type of testing we focused on.

5.3 Paper 3

Paper 3 discusses the first application of RT, ART, and GA for the purpose of RTES system testing based on environment models. The strategies were evaluated on an industrial case study and three artificial problems. Based on the results of the empirical study, we also provided practical guidelines to apply the three testing techniques.

A test case in our context is the setting of simulator generated based on the models. This setting provides values to non-deterministic options of the environment models (e.g., when a sensor should fail). RT is the simplest technique to implement and it randomly selects the values for a new test case. For ART, the paper proposes the use of a specialized distance function. The distance function is used to select a new test case by calculating its distance from previously executed test cases. For SBT, a new fitness function was proposed based on an existing fitness function for model-based testing. The novel fitness function made use of three heuristics: approach level, branch distance, and time distance. Approach level suggests how far the executed test case was to reach an error state (i.e., a state reached when the SUT is faulty). Branch distance suggests that how far was the executed test case

to satisfy a guard, and time distance suggests the distance of the executed test case to trigger a time transition.

The results of the empirical study suggested that no test strategy generally dominates the others. GA was found to be statistically better on one problem, but worst on the other two problems. RT is best on the second problem and ART is better on the third one. On the industrial case study ART showed the best performance.

5.4 Paper 4

Paper 4 improves the fitness function for search-based testing discussed in Paper 3 and empirically evaluates the performance of the improvements. The empirical study is carried out on thirteen artificial problems and one industrial case study. The artificial problems were designed in a way to alter various environment model characteristics in order to evaluate their impact on the search algorithms. Four new heuristics were defined for the fitness functions and they were evaluated individually and in combination. Two search algorithms are evaluated in the paper: GA and (1+1) EA, whereas RT is used as a baseline for comparison.

The first heuristic was improved time distance (ITD) that improved the way time distance was calculated earlier. If a transition should be taken after z time units, but it is not, we calculate the maximum consecutive time c the component stayed in the source state of this transition (e.g., *State2* in the dummy example). To guide the search, we use the following heuristic: $T = z - c$, where $c \leq z$. Earlier the branch distance was calculated after an event was triggered. This mechanism worked fine for transitions other than time transitions, because reducing time distance was not useful when a guard is not satisfied. This heuristic introduces the concept of a look-ahead branch distance, which represents the branch distance of OCL guard on a time transition when it is not fired (i.e., the timeout did not occur). The second heuristic discussed in this paper is “time in risky state” (TIR). TIR favors the test cases that spent more time in the state adjacent to the error state (i.e., the *risky* state). The motivation behind this heuristic is that, the more time spent in a risky state, the higher the chances of events happening in the environment or SUT that lead to the error state (e.g., receive a signal from the SUT).

The third heuristic proposed is “risky state count” (RSC). RSC favors the test cases that enter a risky state more often than those that do so less often. The motivation is similar to that of TIR, that is, to remain in risky state for as long as possible to increase the chances of transitioning to the error state. Finally, the fourth heuristic proposed in the paper is

“coverage” (COV). COV favors the test cases that cover more environment states. The idea behind this heuristic is to increase the coverage of the environment models when the approach level, branch distance and time distance can no longer be improved. The assumption is that having higher environment coverage will result in more diversity in the test cases, which might lead to situations that help reach the error state.

The paper answers the following research questions:

RQ1: What is the effect on fault detection of new order functions having each one of the proposed heuristics: Improved Time Distance (ITD), Time In Risky State (TIR), Risky State Count (RSC), and Coverage (COV) compared to the previously defined basic fitness function for GA and (1+1) EA?

The results showed that ITD with (1+1) EA yields significantly better results for two of the artificial problems. In other cases the performance of the algorithm was the same as that for the basic algorithm. ITD relies on information regarding guarded time transitions in the models. Among the thirteen artificial problems, four did not have any guard or time transition leading to the error state. Even in these cases, ITD shows similar performance to basic fitness with no significant drawbacks.

When TIR was used with GA, it gave significantly better results in two of the artificial problems and was worse in one problem. For other problems, the results of the two algorithms were comparable. When TIR was used with (1+1) EA, it gave significantly better results for five of the 13 artificial problems. In other cases there were no significant differences. Hence the use of TIR in the order function seems to be an effective option.

When RSC was used with GA, it gave significantly better results in one of the artificial problems and showed no significant difference for the other artificial problems. When RSC was used with (1+1) EA, it gave significantly better results for one artificial problem, worse results for another one (AP11), and no statistical differences otherwise. RSC depends on the presence of a loop back to a risky state. Five of the problems had a loop back to the risky state. For all the problems that have a loop to risky states, RSC heuristic performed significantly better or similar to the basic fitness function. But for the problems without such a loop, it can negatively affect performance. When COV was used with GA, there were no statistical differences between the results. When it was used with (1+1) EA, it gave significantly worse results for four of the artificial problems and yielded no significant differences in other cases.

RQ2: Which combinations of the proposed heuristics are best in terms of fault detection?

When the heuristics were executed in combination, we had a total of 16 possible functions for each search algorithm. Overall, based on the results, (1+1) EA with TIR proved to be the best algorithm for both Basic and ITD versions of the heuristic. Based on the results, we concluded that in general search-based algorithms perform significantly worse than RT for the artificial problems where reaching a risky state in the environment model is trivial. If we exclude the results of such artificial problems, then in all the other problems, (1+1) EA with ITD and TIR performed significantly better than other combinations.

RQ3: Between the two search-based algorithms, GA and (1+1) EA, which one works better in terms of fault detection with the new heuristics?

According to the results of experiments, (1+1) EA seems overall to perform significantly better with various combinations when compared to GA using the same combinations of heuristics. An exception to this is when EA is used with the coverage heuristic, in which case it performs significantly worse than GA. Even for the problems with non-trivial approach to risky state, the performance of most of the heuristic combinations for EA is significantly better than their performance with GA. Hence, we can conclude that the fault detection effectiveness of (1+1) EA is higher than that of GA for the kind RTES system testing we focus on.

RQ4: How do the search-based algorithms compare to random testing (RT)?

According to the results of experiments, for simple problems (i.e., where the average success rate of all the algorithms is high or the approach level is trivial) RT performs significantly better than both search-algorithms, but for more difficult problems (i.e., lower success rates or non-trivial approach level), search algorithms perform significantly better. The best technique (1+1) EA-ITD-TIR has an average success rate of 73% for the 13 problems with an average number of 222 test case executions to find a fault.

RQ5: How does the best combination of the proposed heuristics compare to RT and GA and (1+1) EA with basic fitness on the industrial case study?

On the industrial case study, the best combination of proposed heuristics, i.e., (1+1) EA-ITD-TIR, shows significantly better performance over both GA and (1+1) EA. When compared to RT, there is no significant statistical difference, though the combination has

relatively lower success rate (80% compared to 100% for RT). The better performance of RT can be explained by the fact that in the industrial case study, the approach level to risky state was trivial.

5.5 Paper 5

In Paper 5, we combined (1+1) EA with ART to improve the overall performance of our test strategy. The performance of these two algorithms individually is highly dependent on the characteristics of the problem (as suggested by results of Paper 4). In this paper, we proposed a way of combining the strengths of these two algorithms in a way that the dependence on the specifics of the problem is reduced. The proposed hybrid strategy (HS) starts by applying (1+1) EA. If (1+1) EA does not find fitter test cases after running n number of test cases, the testing algorithm is switched to ART. All the test cases that were executed so far are now used for distance calculations in ART. The idea behind switching from (1+1) EA to ART is that there is not enough time for a random walk to get out of a fitness plateau. And so, in this scenario, applying ART can yield better results. Running system test cases is very time consuming, so only few fitness evaluations are feasible within reasonable time (e.g., 1000 test cases can already take several hours). Therefore, in case of fitness plateau, it is reasonable to switch strategy, and rather reward diversity instead of the fitness value. Though the choice of n is arbitrary it can have significant consequences on the performance of this strategy. The best choice for n is also evaluated in the paper.

We conducted an empirical study involving an industrial case study and thirteen artificial problems to answer the following research questions in the paper:

RQ1. Which configuration is best in terms of fault detection for the proposed hybrid strategy (HS)?

According to the results of the empirical study, using a very low (< 50) or very high value (≥ 200) of n causes a degraded performance for HS. With a low value of n , HS makes the switch from (1+1) EA to ART too early, which does not give sufficient time for (1+1) EA to converge and hence running HS becomes similar to only running ART. In cases where ART performs well, such configurations of HS also perform well. Similarly, when HS switches too late, it does not give enough time to ART and hence running HS is similar to running (1+1) EA in such cases. These configurations perform well in cases where (1+1) EA performs well and poor otherwise. The best results are provided for values between 50 and 100 and the differences in results in this range are not significant. Though

the results are not fully consistent across all problems, configuration $n = 50$ has the best average rank across all problems and is always very close to the maximum success rates.

RQ2. How the fault detection of the best HS configuration compares with the performance of ART, (1+1) EA, and RT for (a) the artificial problems and (b) the industrial case study (IC)?

Based on the results, HS shows significantly better performance in terms of fault detection (an overall 88% success rate for artificial problems and 100% for the industrial case study) than the other three algorithms (for artificial problems: ART: 63%, RT: 64%, and (1+1) EA: 74% and for the industrial case study: ART: 100%, RT, 97%, (1+1) EA: 74%. Unlike the other strategies, variations in environment properties do not have a drastic impact on the performance of HS and it is therefore the most practical approach, showing consistently good results for different problems.

5.6 Paper 6

For successful model-driven engineering (MDE) applications, a comprehensive methodology for modeling should be adopted that is specific to the problem being solved and adequate for the application domain. This paper discusses our experiences of applying the Unified Modeling Language (UML) and the UML profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) to solve three distinct industrial problems related to the use of real-time embedded systems (RTES). The work discussed in this thesis, environment model-based testing of RTES, is one of the addressed problems. The common experiences from these three problems are merged and summarized into a framework to guide future industrial applications of UML/MARTE. The framework provides a set of detailed guidelines on how to apply MARTE in industrial contexts and will help future modelers reduce the gap between the modeling standards and industrial needs.

The proposed framework consists of six high level steps that are derived based on our experience. The first step is a domain analysis of the industrial context in order to understand the domain and the problem. The second step consists of identifying the proper set of notations for modeling. UML and MARTE are both international modeling standards and cater the needs of a large variety of problems and domains. To apply them in a particular context, identifying a relevant subset of UML/MARTE is a very important task. The next step is to provide extensions to UML/MARTE according to the requirements of the domain and problem being handled in the form of a profile. Selection of modeling tools

can also greatly impact the success of industrial application in a later stage and this forms the third step of the framework. Some of the important factors to consider are the cost of the tool, its supported technologies, and usability of the tool in modeling the selected subset of UML/MARTE. For a successful application of UML/MARTE, only selecting a set of notations is not sufficient, rather we also need to define a set of guidelines on how to use these notations to achieve the goals (for example, as we provided in Paper 1 and Paper 2). This forms the fourth step of the framework. Finally, as a last step of the framework, we provide guidelines on how to actually apply UML/MARTE in industrial contexts (e.g., by conducting live modeling sessions).

5.7 Paper 7

This paper devises novel search heuristics to solve OCL constraints for test data generation. We evaluated two search-algorithms, GA and (1+1) EA, and used RT as a comparison baseline. A search-based OCL constraint solver was developed based on the heuristics and evaluated on an industrial case study. The heuristics are designed for various elements of OCL expressions, including operations on primitive types and collections. These heuristics are then evaluated on an industrial case study of a Video Conferencing Software developed by Cisco Systems. The following research questions were answered in this paper:

RQ1: Are search-based techniques effective and efficient at solving OCL constraints in the models of industrial systems?

The results show that (1+1) EA outperformed both RS and GA, whereas GA outperformed RS. We observed that, with an upper limit of 2000 iterations, (1+1) EA achieves a median success rate of 80% but GA did not exceed a median roughly 60%. The success rates for (1+1) EA were above 50% and most of them were close to 100%. Constraints with the lowest success rates were seven and eight clauses long.

RQ2: Among the considered search algorithms, which one performs best in solving OCL constraints?

According to the results of the empirical study, there is strong evidence to claim that (1+1) EA is more successful than both GA and RT. (1+1) EA was not only successful in solving the constraints with more frequency, but the magnitude of difference with the other two strategies was also large.

5.8 Paper 8

The paper is a journal extension of Paper 7 with the following differences:

1. Additional heuristics have been added in the paper such as heuristics for operations on collections, special operations (e.g., oclInState), and user-defined operations.
2. The empirical evaluation based on the industrial case study has been improved in the following ways:
 - a. The case study is extended with new constraints
 - b. An additional search algorithm, Alternating Variable Method (AVM), is included
3. The empirical evaluation of the individual heuristics on several artificial problems has been added.
4. The empirical evaluation comparing our work with an existing work has been extended. The evaluation is based on the industrial case study.

The paper discusses the following research questions:

RQ1: Are search-based techniques effective and efficient at solving OCL constraints in industrial system models?

The results show that AVM outperformed all the other three algorithms, i.e., (1+1) EA, RS, and GA. AVM also achieved 100% success rate (i.e., number of times it was able to solve a constraint) compared to 98% of (1+1) EA, 65% of GA, and 49% of RT. This showed that search-based techniques, specifically AVM and (1+1) EA are effective and efficient in solving constraints for industrial models.

RQ2: Among the considered search algorithms (AVM, GA, (1+1) EA), which one fares best in solving OCL constraints and how do they compare to RT?

The results indicate that among the three search algorithms, AVM had highest success rate, followed by (1+1) EA. GA showed relatively lower success rates. RT in comparison to these algorithms showed lowest success rate.

RQ3: Does the optimized branch distance calculation improve the effectiveness of search over non-optimized branch distance calculation?

When AVM and (1+1) EA with fitness function using optimized branch distance were compared with the ones with fitness function using non-optimized branch distance, the results showed that for both the algorithms, optimized branch distance showed significantly better results. In cases where there were no differences in success rates, algorithms with optimized branch distance took significantly less iterations to solve the problems.

6 Future Directions

Regarding future work, a first step to carry out is an empirical cost-benefit analysis of the proposed model-based testing approach. The cost of building and modifying the environment models needs to be compared with that of the manual changes to simulators and test suites. Intuitively, the latter should be much larger than the former, but it nevertheless should be investigated. Estimates of the cost of field failures need to be considered as well to obtain more reliable and complete comparisons of cost-effectiveness among test strategies.

Since our testing approach was based on the needs of our industry partners, we only focused on real-time systems with relatively soft deadlines of hundreds of milliseconds. A possible research direction is to adapt the approach for systems with strict and shorter time deadlines. For this purpose, we will need to investigate the simulator generation for other languages and specific platforms (e.g., C).

The work reported in this thesis is restricted to one randomly generated configuration of the environment structure. Another research direction is to analyze how to properly use the domain models for effective automated testing of different configurations of the RTES environment. Strategies can be investigated to generate configurations at run time in a way that increases the effectiveness of testing algorithms.

7 Conclusion

Black-box system testing of Real-time Embedded Systems (RTES) on their development platforms is required to verify the correctness of these systems without involving the deployed hardware and other physical components of their environments. This approach typically involves simulations of the behavior of environment components in a way that is transparent to the RTES. Such a strategy allows early and fully automated system testing, even when the hardware is not yet available. It is also helpful in situations where testing RTES for critical failures in their actual environments is either not feasible, too costly, or might have catastrophic consequences.

This thesis reports on a model-driven, automated black-box system testing strategy for real-time embedded systems (RTES) based on their environments. The strategy is developed while keeping in consideration the practical requirements of two industrial partners that are, we believe, representative of a wider category of RTES developers. We purposefully took a practical angle and our approach does not require software engineers to use additional, specific notations for simulation and testing purposes, but only involves slight extensions of existing software modeling standards and a specific modeling methodology. First we developed a precise methodology for environment modeling of RTES. The methodology is based on standards: UML, MARTE profile and OCL for modeling the structure, behavior, and constraints of the environment. We, and this is part of our methodology, made a conscious effort to minimize the notation subset used from these standards. Our modeling methodology entails the use of constructs (e.g., non-determinism, error states, and failure states), which are essential to enable fully automated system testing (i.e., choice, execution and evaluation of the test cases). We modeled the environment of three artificial problems and two industrial RTES in order to investigate whether our methodology and the notation subsets selected were sufficient to fully address the need for automated system testing. Our experiences showed that this was the case. Lessons learned from industrial applications of the methodology were also summarized to guide future practitioners.

Secondly, based on a careful analysis of the literature, we concluded that none of the existing code generation approaches in the literature address the constructs required to support the testing of RTES through environment simulation. We implemented the code generation rules for the simulator using model-to-text transformations with MOFScript, thus producing a set of Java classes. Our empirical evaluation, based on our

five case studies, shows that the developed rules are sufficient and that they are correct as far as fault detection is concerned. The automated simulator generation is expected to save a significant amount of effort, although empirical studies in industrial contexts will be necessary to support such a claim with increased confidence. By using our environment models and the generated simulators, it was possible to automatically find new, critical faults in one of the industrial case studies using fully automated, random and search-based testing.

The third part of the thesis concerned OCL constraint solving, for which, we defined search heuristics involving branch distance functions for various types of expressions in OCL to guide the search algorithms. We demonstrated the effectiveness and efficiency of our search-based constraint solver to generate test data in the context of the model-based, robustness testing of an industrial case study of a video conferencing system. Even for the most difficult constraints, with research prototypes and no parallel computations, we obtained test data within 2.96 seconds on average.

Last but not least, we discussed various strategies for test case generation based on environment models. We defined and iteratively improved fitness functions for search-based algorithms. We also evaluated the use of adaptive random testing (ART) and random testing (RT) in our context. The experiments were conducted on an industrial case study and a number of artificially created problems with varying properties. Based on the results of initial experiments, we proposed a hybrid strategy (HS) for testing that combined (1+1) Evolutionary Algorithm (EA) and ART. The strategy was developed to combine the benefits of both algorithms, since their individual results varied greatly depending on the failure rate of the system under test and the properties of its environment. The ultimate goal was to obtain a strategy with consistently good results. Overall, the results indicate that HS shows significantly better performance in terms of fault detection (an overall 88% success rate for artificial problems and 100% for the industrial case study) than the other three algorithms (for artificial problems: ART: 63%, RT: 64%, and (1+1) EA: 74% and for the industrial case study: ART: 100%, RT, 97%, (1+1) EA: 74%). Unlike the other strategies, variations in environment properties do not have a drastic impact on the performance of HS and it is therefore the most practical approach, showing consistently good results for different problems.

8 References for the Summary

- [1] Artemis. (2011, June 13, 2011). *Artemis Joint Undertaking - The public private partnership for R & D Embedded Systems*. Available: http://artemis-ju.eu/embedded_systems
- [2] B. M. Broekman and E. Notenboom, *Testing Embedded Software*: Addison-Wesley Co., Inc., 2003.
- [3] *WesternGeco*. Available: <http://www.slb.com/services/westerngeco.aspx>
- [4] *Tomra AS*. Available: <http://www.tomra.no>
- [5] OMG, "Unified Modeling Language Superstructure, Version 2.3, <http://www.omg.org/spec/UML/2.3/>," ed, 2010.
- [6] OMG, "Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0, <http://www.omg.org/spec/MARTE/1.0/>," ed, 2009.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, 1995.
- [8] P. M. Kruse, J. Wegener, and S. Wappler, "A highly configurable test system for evolutionary black-box testing of embedded systems," presented at the Proceedings of the 11th Annual conference on Genetic and evolutionary computation, Montreal, Canada, 2009.
- [9] F. Lindlar, A. Windisch, and J. Wegener, "Integrating Model-Based Testing with Evolutionary Functional Testing," presented at the Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, 2010.
- [10] F. Lindlar and A. Windisch, "A Search-Based Approach to Functional Hardware-in-the-Loop Testing," presented at the Proceedings of the 2nd International Symposium on Search Based Software Engineering, 2010.
- [11] M. Short and M. J. Pont, "Assessment of high-integrity embedded automotive control systems using hardware in the loop simulation," *J. Syst. Softw.*, vol. 81, pp. 1163-1183, 2008.
- [12] G. Francis, R. Burgos, P. Rodriguez, F. Wang, D. Boroyevich, R. Liu, and A. Monti, "Virtual Prototyping of Universal Control Architecture Systems by means of Processor in the Loop Technology " presented at the Twenty Second Annual IEEE Applied Power Electronics Conference, APEC 2007 - Twenty Second Annual IEEE 2007
- [13] (2010). *Object Constraint Language Specification, Version 2.2*. Available: <http://www.omg.org/spec/OCL/2.2/>
- [14] (2010). *UML Profile for Schedulability, Performance and Time*. Available: http://www.omg.org/technology/documents/profile_catalog.htm
- [15] M. Harman, S. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," Department of Computer Science, King's College London, TR-09-032009.
- [16] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," *IEEE Transactions on Software Engineering*, vol. 99, 2009.
- [17] T. Chen, F. Kuo, R. Merkel, and T. Tse, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, pp. 60-66, 2010.
- [18] M. Samek, *Practical UML statecharts in C/C++: event-driven programming for embedded systems*: Newnes, 2009.
- [19] L. Ferreira and C. Rubira, "The reflective state pattern," presented at the Proceedings of the Pattern Languages of Program Design, Monticello, Illinois-USA, 1998.

- [20] J. Oldevik, "MOFScript user guide," *Version 0.6 (MOFScript v 1.1. 11)*, 2006.
- [21] P. McMinn, "Search-based Software Test Data Generation: A Survey," *Software Testing Verification and Reliability*, vol. 14, pp. 105-156, 2004.
- [22] Cisco Inc. Available: <http://www.cisco.com>

Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies

Muhammad Zohaib Iqbal, Andrea Arcuri, Lionel Briand

In: Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS), Model Driven Engineering Languages and Systems. Springer Berlin / Heidelberg, 2010, pp. 286-300

Abstract—The behavior of real-time embedded systems (RTES) is driven by their environment. Independent system test teams normally focus on black-box testing as they have typically no easy access to precise design information. Black-box testing in this context is mostly about selecting test scenarios that are more likely to lead to unsafe situations in the environment. Our Model-Based Testing (MBT) methodology explicitly models key properties of the environment, its interactions with the RTES, and potentially unsafe situations triggered by failures of the RTES under test. Though environment modeling is not new, we propose a precise methodology fitting our specific purpose, based on a language that is familiar to software testers, that is the UML and its extensions, as opposed to technologies geared towards simulating natural phenomena. Furthermore, in our context, simulation should only be concerned with what is visible to the RTES under test. Our methodology, focused on black-box MBT, was assessed on two industrial case studies. We show how the models are used to fully automate black-box testing using search-based test case generation techniques and the generation of code simulating the environment.

1. Introduction

Real-Time Embedded Systems (RTES) are largely used in critical domains where high system dependability is required and expected. The basic characteristic of RTES is that they react to external events within certain time constraints. Extensive testing of such systems is important in order to verify their correct behavior under different timing constraints and adverse situations of the environment (or context). It is also important to

verify that the system under test (SUT) does not lead the environment to a hazardous state. Testing RTES is particularly challenging since they operate in a physical environment composed of possibly large numbers of sensors and actuators. There is usually a great number and variety of stimuli with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. Testing all possible sequences of stimuli/events is not feasible. Hence, systematic testing strategies that have high fault revealing power must be devised. Manually writing appropriate test cases for such complex systems would be a far too challenging and time consuming task. If any part of the specification of the RTES changes during its development, a very common occurrence in practice, then the expected output of many test cases would potentially need to be recalculated manually. Automated test-generation and the use of an automated oracle are essential requirements when dealing with complex industrial RTES.

Moreover, testing the RTES in the real environment usually entail a very high cost and in some cases the consequences of failures would not be acceptable, for example when leading to serious equipment damages or safety concerns. In many cases the hardware, e.g., sensors and actuators, is not yet available at the time of testing as software and hardware are typically developed concurrently in RTES development. Since testing RTES on the real environment is not a viable solution, the use of a simulator is a common alternative.

In our work, we address the above issues by devising a comprehensive, practical methodology for black-box, model-based testing (MBT). The main contributions of this paper are as follows: It provides an environment modeling methodology based on industrial standards and targeted at MBT, and evaluates it on two industrial case studies. The models describe both the structural and behavioral properties of the environment. Given an appropriate level of detail, defined by our methodology, they enable the automatic generation of the environment simulator. The models can also be used to generate automated test oracles. These could, for example, be invariants and error states that should never be reached by the environment during the execution of a test case. Moreover, the models can further be used to automatically choose test cases. Sophisticated heuristics to choose appropriate test cases are automatically derived from the models without any intervention of the tester. To summarize, the only required artifacts to be developed by testers is the environment model and the rest of the process is expected to be

fully automated. By using this automated MBT technology, one of our industrial partners was able to find new critical faults in their RTES. This paper focuses on how to make environment modeling as easy as possible for the purpose of supporting black-box, MBT, and shows its use for test automation. Due to space constraints, we only briefly discuss the details for code generation.

To support environment modeling in a practical fashion, we have selected standard and widely accepted notation for modeling software systems, the UML and its standard extensions. We use the MARTE [1] extensions for modeling real-time features and OCL for specifying constraints. We have also provided lightweight extension to UML to make it more useful in our context. As we will discuss later, environment modeling is not a new concept. But, most of the approaches use non-standardized notations or grammars for modeling, which makes them difficult to apply from a practical standpoint. To the best of our knowledge, modeling the environment of industrial RTES systems using a combination of UML, MARTE, and OCL has not been addressed in the literature. By using the proposed methodology, the software testers (who are primarily software engineers) can model the environment with a notation that they are familiar with and at a level of precision required to support automated MBT.

The importance of selecting standards for modeling was highlighted by the application of methodology on the two industrial case studies that belonged to completely different domains. An alternative to using standard notations for modeling could have been to create a Domain Specific Language (DSL) for environment modeling. Since the methodology needed to be generic for RTES irrespective of their application domain, making a DSL was not feasible. Making a DSL would have also reduced the benefits that we obtained from using standards and could have only been justified if existing standards did not fit our needs. Our case studies were developed using Enterprise Architect and IBM Rational Software Architect, though any of the widely available UML tools could have been used for this purpose.

The rest of the paper is organized as follow. Section 2 discusses the related work on environment modeling and testing based on environment models. The environment modeling methodology and simulation is discussed in Section 3. Section 4 describes the use of the environment modeling methodology for automated testing. Section 5 discusses

the case studies on which the methodology was applied on and finally Section 6 concludes the paper.

2. Related Work

There are a few approaches reported in the literature for the environment modeling of embedded systems. Kishi and Noda [2] present an approach for modeling the environment of an embedded system using an aspect-oriented modeling technique. Karsai et al. [3] propose a new language for modeling the environment of an embedded system. Choi et al. [4] use annotated UML class and sequence diagrams for modeling and simulation of environment. Kreiner et al. [5] present a process to develop environment models for simulation of automatic logistic systems and its environment. Axelsson [6] evaluates how UML can be used to model real-time features and provides extension to UML for modeling of real-time systems and their environments. Gomaa [7] discusses the use of a context diagram for modeling the relationship between an RTES and its external entities. Friedentahl et al. use the concept of SysML block diagram and activity diagrams to represent the system and its interfaces with environment components [8].

There are a few works reported in literature that discuss testing based on the environment of a system. Auguston *et al.* [9] discuss the development of environment behavioral models using Attributed Event Grammar for testing of RTES. Bousquet *et al.* [10] present an approach for testing of synchronous reactive software by representing the environmental constraints using temporal logic. Larsen *et al.* [11] propose an approach for online testing of RTES based on time automata and environmental constraints. Heisel *et al.* [12] propose the use of a requirement model and an environment model using UML state machines along with the model of the SUT for testing. Adjir *et al.* [13] discuss a technique for testing RTES based on the model of the system and model of intended assumptions in the environment in Labeled Prioritized Timed Petri Nets.

As discussed above, there are approaches in literature that deal with modeling the environment of a system for various purposes. Most of these approaches are only limited to modeling the static structure of the environment, as they do not focus on test automation. The approaches that deal with modeling of behavioral aspects either use notations with which the software engineers are not familiar, or provide extensions for environment modeling that do not have well-defined semantics. Moreover, the properties of the

environment, such as its timeliness and non-determinism, are not modeled in a standard way. The environment models should be compatible with other standard techniques available for model manipulation, e.g., model transformations, consistency checking. For this reason, the modeling language should have well-defined constructs. All environment modeling approaches aimed at supporting testing, except by Heisel *et al.* [12], use non-standard languages for modeling. Heisel *et al.* models both the SUT and the environment, which does not fit our purpose: black-box, system testing. Moreover, they model the concepts of probabilities and time using non-standard notations, without using the UML extension mechanisms. Last but not least, none of the relevant work assesses their environmental methodology on an actual RTES system, which we believe is a requirement to assess the credibility and applicability of any MBT approach.

3. Environment Modeling - Methodology

If environment models are to be used for RTES, they should not only be sufficiently detailed, but should also be easy to understand and modify as the environment and RTES evolve. To handle the complexity of realistic RTES environments, the modeling language should have provision for modeling at various levels of abstraction. The modeling language should also have well-defined syntax and semantics for the tools to analyze the models and for the humans to accurately understand them. The language should also provide features (or allow possible extensions) for modeling real world concepts, real-time features, and other concepts, such as non-determinism, required by the environment components. The UML, MARTE profile, and the OCL together fulfill the important requirements of an environment modeling language.

Even though we are using the same notations to model the environment that are used for modeling software systems, it is important to note that the methodology for environment modeling is significantly different from system modeling. While modeling for the industrial cases, we abstracted the functional details of the environment components to an extent that only the details visible to the SUT were included. For environment behavior modeling, non-determinism is widely used, which is not nearly as common when modeling the internal behavior of a system.

For testing the system based on its environment, the behavior details of the environment are as important as its structural details. Structural details of the RTES environment are

important to understand the overall composition of the environment (e.g., number and configuration of sensors/actuators), the characteristics of various components, and their relationships. We choose to model these details in the form of a *Domain Model* developed using UML class diagrams. The behavioral details of environment components are required to specify the dynamic aspects of the environment, for example, to determine the possible environment states, before and after its interactions with the SUT, and to specify the possible interactions between the SUT and its environment. For behavioral details, we used the UML State Machines augmented with the MARTE profile.

In the following subsections, we discuss the methodology for modeling the environment of a RTES. We also discuss various guidelines based on our experience of applying the methodology on two industrial case studies.

3.1. Modeling Structural Details as Environment Domain Model

The environment domain model provides information of the components of the environment, their characteristics, their relationships with one another and the SUT, and information regarding signal sending and reception. The various components modeled in the domain model together form the overall environment of the SUT. This means that all these components (their instances) will run in parallel with each other. Each component in the domain model can have a number of instances in the RTES environment. The information about the number of possible instances of a component in the environment is modeled as cardinalities on the associations between different components in the domain model. Therefore, the domain model can be used to obtain a number of potential configurations of the environment. Fig. 1 shows the partial domain model for the environment of one of our industrial cases, the sorting machine (named as *SortingBoard* in the figure). The sorting machine is part of an automated bottle recycling system and further details of the case study can be found in Section 5. The model shows various motors, sensors, mechanical devices taking part in sorting, and other systems the *SortingBoard* communicates with.

Note that the domain model that we develop is different from the ones commonly discussed in literature (e.g., [14]). The components represented as classes in the environment domain model will not necessarily relate to software classes. They may correspond to systems, users and concepts related to various natural phenomena. Domain

modeling here is not a starting point for software analysis. The identification of components in the domain model, their properties, and their relationships is also different from what is commonly done for software analysis. Following, we further discuss various guidelines for modeling the structural details of a RTES environment.

3.1.1. Environment Components to be Included.

Initially, all the environment components that are directly interacting with the SUT are included in the domain model. Then, each of these components is further refined to a level where we are certain to cover the important details for simulating the environment needed to test the SUT. If at any time the behavior of an environment component was getting too complex, when possible, we decomposed the component and divided its behavior into multiple concurrent state machines. This is especially useful if a component can be divided into components that are similar to existing components, so that we can specialize existing state machines. We used the stereotype <<*context*>> to represent components of the environment in the domain model. The components of the environment are made to communicate with each other and the SUT through signals, and are modeled as active objects.

3.1.2. Relationships to be Included.

All those associations representing the physical or logical relationships among various environment components, or that were needed for components to communicate, should be included. A number of components in the environment might be similar to each other (e.g., various types of sensors). It is useful to relate these components (and their behavior) using the generalization/specialization relationship for simplifying the model, as our experience shows that such domain models get highly complex. For example, in the sorting machine case study, we modeled the association of the *SortingBoard* with the *SortingArm*, which is controlled by the board, and the *ItemSensor* that reports arrival of an *Item* (e.g., bottle). We used generalization in multiple places, including motors and sensors as shown in Fig. 1.

3.1.3. Properties to be Included.

From all properties that may characterize environment components, it is important to include only those properties that are visible to the SUT (or have an impact on a

component that is visible to the SUT). These may include attributes that have a relationship to the inputs of the SUT, that constrain the behavior of a component with respect to the SUT, or that contribute to the state invariant of a component that is relevant to the SUT. In Fig. 1, all the modeled properties of *Item* are either visible to the *SortingBoard* or are used by other components. For example, the *serialNum* and *materialType* of *Item* is assigned by *VendingMachine* and is used by the *SortingBoard*.

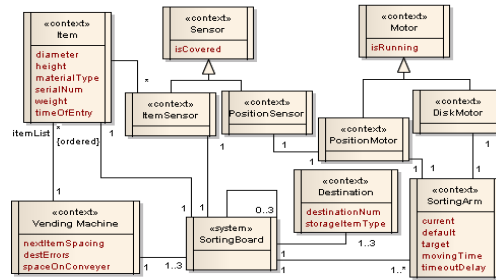


Fig. 1. Partial environment domain model showing properties and relationships of the sorting machine case study

3.1.4. Modeling the SUT.

It is important to include the SUT in the environment domain model, so that its relationship with the other environment components can be specified. It is also useful to include the details of signal receptions by the SUT from other environment components. The SUT is stereotyped as `<<system>>`. The stereotype was used initially by Gomaa [7] to refer the system in a context diagram. The SUT modeled in the domain model should represent the SUT and its execution platform, as a single component.

3.2. Modeling Behavioral Details with UML State Machines & MARTE

For modeling the behavior details of the environment that have an impact on the SUT, we developed the UML State Machines with MARTE real-time extensions for various components in the environment. As discussed earlier, the environment components run in parallel to form the environment of the RTES. The components can send signals to each other and to the SUT. We can also view the environment as having one state machine with orthogonal regions, one for each component. Fig. 2 shows the state machine of a component for one of the industrial case studies. We have abstracted out the concepts for

confidentiality reasons. Following, we discuss the details of the methodological guidelines we followed.

3.2.1. Identifying Stateful Components.

Components whose states either affect the SUT or are affected by the SUT should be modeled with state machines. Apart from these components, it is also useful to model the behavior of other components on which we would like control during the simulation.

Overall, the environment should be modeled in a way that enables, after the initial configuration and provision of input data (parameters and guards), the full simulation of the interactions with the SUT. All the context components shown in Fig. 1 are stateful components of the sorting machine case study. For example, the *SortingArm* component was modeled as stateful since it receives signals from the *SortingBoard* and reacts differently based on its current state.

3.2.2. States to be Included.

It is important to determine the right level of abstraction for a component state machine. If we want to precisely model the behavior of an environment component, this might lead to a large number of states. We are, however, only interested in state changes that have an impact on the SUT. A single state in an environment model state machine may correspond to a large number of concrete or physical states. For example, in the sorting machine, the Item states that were modeled were all related to its movement through the sorting machine whereas its other possible states were not of interest as an environment component of the *SortingBoard*.

3.2.3. Modeling Users in the Environment.

Generally, for software system modeling users are only modeled as sources of inputs and data. The behaviors of users with respect to the system are mostly not considered. In the environment modeling methodology, it is useful to model the behavior of users in the environment to have a control over the inputs/outputs of the various components or the SUT. If a user participates in multiple roles, it is useful to model each role a user plays as a separate component. In the sorting machine case study, we modeled two different users (the operator and the persons who enter the items for sorting), each of them had

considerable non-deterministic behavior. In certain cases it can be interesting to model both the expected and unexpected behavior of users using the proposed methodology.

3.2.4. Modeling Abstract Phenomena.

Sometimes it is necessary to model abstract physical concepts, such as temperature, heat, voltage, and current. Mostly, information regarding these phenomena can be obtained and controlled through sensors and controllers, such as a temperature controller or sensor. Modeling of such concepts explicitly as environment components can be useful if a change in the state of these concepts impacts multiple components simultaneously, or if it is not possible to identify a related component in the environment that can act as a controller or sensor of this concept for simulation. As an example, consider a RTES on a vehicle that indicates its driver the time for a pit stop. The tires of a vehicle can burst when the temperature of the road gets too high. If there is no sensing mechanism available in the environment, then it is useful to make a state machine of temperature, with possibly two states representing below and above danger temperatures.

3.2.5. Modeling Transitions & Action Durations.

Most of the transitions in the state machines of the components will either be based on signal events or time events. Timeout transitions are an important concept in RTES environment models. The MARTE *TimedEvent* concept is used to model timeout transitions, so that it is possible for them to explicitly specify a clock. Each environment component may have its own clock or multiple components may share the same clock for absolute timing. The clocks are modeled using the MARTE's concept of clocks. Specifying a threshold time for an action execution or for a component to remain in a state is possible using the MARTE *TimedProcessing* concept. This is also a useful concept and can be used, for example, to model the behavior of an environment component when the RTES expects a response from it within a time threshold. When a *SortingArm* is signaled to move, after staying some time in the *Moving* state, it transitions to the *Not Moving* state (see Fig. 2).

3.2.6. Modeling Non-Determinism.

Non-determinism is a particularly important concept for environment modeling and is one

of the fundamental differences between models for system modeling and models for environment modeling. Following we discuss different types of non-determinism that we have modeled for our case studies.

Specifying exact value for timeout transitions might not always be possible for RTES environment components. To model their behavior in a realistic way, it is often more appropriate to specify a range of values for a possible timeout, rather than an exact value. Moreover, the behavior of humans interacting with the RTES is by definition non-deterministic. For modeling this behavior, we can add an attribute in the environment component and use OCL to constrain the possible set of values of the attribute and then use this attribute as a parameter of a timeout transition. In the sorting machine case study, the *SortingArm* may reach a sorting location from its center between 5 sec and 6 sec, depending on various physical conditions. This is modeled through the attribute *movingTime*, which is passed as a parameter to the change event on the transition from *Moving* to *Not Moving*. Legal values for the attributes are constrained using OCL.

Another important form of non-determinism is to assign probabilities to the transitions of state machines. In an RTES environment, we sometimes only know the probability of a component to go into a particular state over time and we are not sure about the exact occurrence of such conditions. For example, we can say that the probability of a car engine to overheat after running continuously for 10 hours is 0.05, but we cannot be certain about the exact instance in time when this situation will happen. We can model this in the engine state machine with a transition going from *Normal Temperature* state to *Overheated* state, during an interval of 10 hours, with probability of 0.05. For modeling these scenarios, we assigned a probability on the transitions using the property *prob* of the MARTE *GaStep* concept. Whenever a timeout transition has the *gaStep* stereotype applied with a non-zero value of *prob*, the combination will be comprehended as the probability of taking the transition over time of timeout transition. In the sorting machine case study, a *SortingArm* can get stuck in a position (e.g., because of a bottle blocking it or the arm jamming) with a probability 0.02 in a minute if it is not moving and a higher probability when it is moving. This can be modeled as shown in Fig. 2 by the transitions from *Not Moving* and *Moving* to *Sorter Stuck*. The sending of non-deterministic signals can also be modeled using this type of transitions, by placing them in the actions of such transitions.

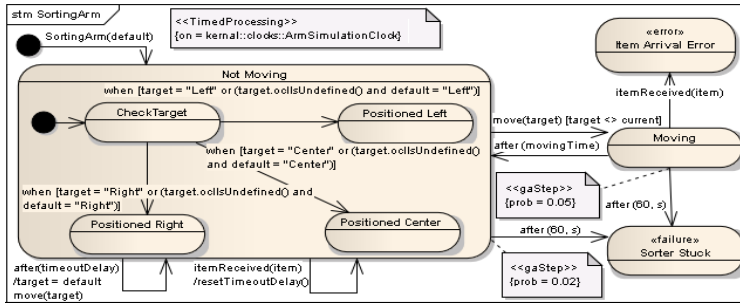


Fig. 2. State Machine of the SortingArm component in the sorting machine case study

Another type of probability that we modeled in our case studies is for the situations where one event can lead to multiple possible scenarios, but all of them are mutually exclusive. For example, we might want to represent the fact that during the communication with the SUT there is a chance that signals are received with or without distortion. To make the models more realistic, we assigned probabilities to each of such scenarios in the environment component. In terms of UML state machines, this means that multiple transitions are outgoing from one state based on the same event (maybe with identical guard). For modeling these scenarios, we assigned the MARTE *gaStep* stereotype to each of the multiple possible outgoing transitions. The example of communication with the SUT can be modeled by having two transitions going out of the environment component state on receiving of a signal, one labeled with a probability that the signal was corrupted and the other with the probability that the signal was fine. Modeling the distribution of event arrivals and timeout transitions can be useful for validation purposes, but is out of the scope of this paper, since our goal is verification of the SUT. Nevertheless, this type of information can be easily expressed in the model using the MARTE profile.

3.2.7. Modeling Error & Failure States.

In the environment models, two types of states play a particularly important role: the *error* states and the *failure* states.

Environment error states are those states that the environment goes into because of unwanted response(s) (or lack of) from the SUT. Every component in the environment may have error states. If any component of the environment reaches one of these error states, then it means that the SUT is faulty. We use the stereotype <<error>> for such states in the environment model. For a *SortingArm*, an *Item* should not arrive while the arm is

moving. This is an error state of the environment and can be caused if arm is not made to move on time by the *SortingBoard*. In Fig. 2, this has been modeled with the *Item Arrival Error* state.

Failure states model possible failures of environment components. A component may fail in several different ways with different consequences for the SUT. The SUT should appropriately behave under known, failing conditions. A failure can happen at any time during the execution of a component, e.g., a sensor may break at any time, and is modeled as non-deterministic behavior (as discussed). We use the stereotype <<*failure*>> for these failure states. The *Sorter Stuck* state discussed earlier, in which the *SortingArm* is stuck and cannot change its position, is a failure state of the environment.

3.3. Modeling the Constraints

To apply constraints on the relationships and restrictions on various value combinations (or state combinations) of objects, we have used the Object Constraint Language (OCL). We have also used OCL for representing the guards on the state machines, various state invariants and general constraints on the relationships of environment components.

RTES environment consists of a number of components including some real-world concepts (e.g., temperature, air pressure). If we consider all the various components of environment together, it is important to restrict the possible state combinations of these components to avoid infeasible situations (e.g., reverse and forward movement of motors is not possible at the same time). In our methodology, we have used OCL to specify constraints for such scenarios. For example, for the sorting machine, if a *SortingArm* is moving then only one *DiskMotor* and *PositionMotor* should be running at a given time. If the arm is not moving, both the motors should not be running. There can be a number of such constraints and it is important to model them to have a realistic simulation and testing based on the models. Otherwise, the models would end up in states that are not practically possible.

State invariants in the environment also play a significant role. Based on the values of the attributes of the component, the state invariants are used to evaluate the current state of the environment and derive state oracles (i.e., is the environment in the expected state?). We have used OCL to specify the state invariants. We also used OCL to specify the overall

set of values that an attribute of an environment component can take. Last, the OCL constraints were also used for modeling non-determinism as discussed earlier.

3.4. Environment Modeling Profile

Our goal was to model the environment based only on the standard UML and its existing extensions as much as possible. We applied the standard notations and based on our needs for those case studies, where required, we provided light weight extensions to UML. In this section we will discuss the subsets of UML and MARTE that we used and the lightweight extensions that we have provided for environment modeling. From a practical standpoint, it was important to identify these subsets for the methodology, since the UML and MARTE standards are very large and most organizations would be reluctant to adopt such large notations.

We used the concept of Context, System, Error, and Failure under the form of UML stereotypes. Context is used to represent an environment component and is applied on the classes of the domain model. Similarly, System is also applied on the classes of the domain model and represents the SUT. Error represents the states of environment component that are only taken if there is an error in the SUT. Failure is also applied on the states and represents a failure in the environment. Within UML, we used the concept of Class diagram, State Machines. From MARTE, we only used the Time package and the GaStep concept from the GQAM package as shown in Fig. 3. This small subset of UML and MARTE was sufficient for modeling our two industrial case studies for the purpose of automated black-box testing.

3.5. Simulation of Environment Models

Due to size constraints, we cannot go into the details of the simulation and only briefly discuss it. The environment models developed using our methodology with UML and the MARTE profile are transformed into a RTES environment simulator in Java using a model to text transformation. The transformation was based on an extended version of the state pattern that accounts for asynchronous communication, time events, and change events. The simulator is used to test a RTES in conditions similar to its real environment. Since the standard for a concrete syntax of the UML Action Language is still not finalized, we made use of Java to specify actions. Once there is a standard UML Action Language, the

actions can be written in that language and then translated into the target language of the RTES. For our case studies, the actions are written in Java and are converted into Java method calls.

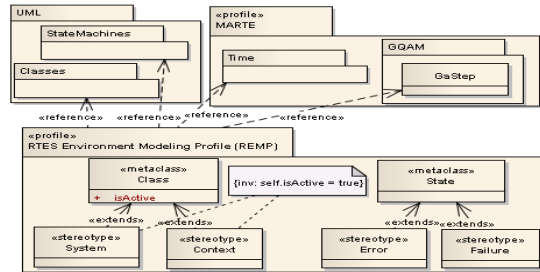


Fig. 3. Profile diagram showing various stereotypes and references

4. Model-based Testing based on Environment Models

In this section we briefly discuss how our modeling methodology is used to achieve automated system testing. Further details can be found in [15].

The UML/MARTE models of the environment are used to automatically generate a simulator for it. The simulator is used to test the RTES on the development platform. The information from the models is used to guide the generation of test cases and for generating automated oracles, which enable fully automated testing. Once test case and oracle generation is completely automated, it is possible to execute and evaluate a large number of test cases.

In our methodology, a test case is the setting used for the simulator. The information of what to configure in the simulator is automatically derived from the models and it is given as input to the test engine. Two types of setting are necessary:

- Number and relations of the environmental components. For example, given a state machine representing a sensor, the Domain Model is used to determine how many sensors can be connected to the RTES (and so, we would know how many running instances we need for that state machine). Several different combinations are possible.
- Each state machine can have non-deterministic events. The models are used to specify them and to provide details of their type. When the simulator is running,

every time it requires a value to calculate a non-deterministic event, it then queries the test engine to obtain such values.

At the current moment, we have not investigated different configurations based on the Domain Models. We have focused on testing the behavior of the RTES given a single configuration. The goal of the testing is to provide a valid setting for the non-deterministic events such that an environmental error state (Section 3.2) is reached during the simulation, if any fault is present.

The simplest testing technique would be to provide (valid) random values each time the simulator queries the test engine for values to use in non-deterministic events. But more sophisticated techniques that exploit the information in the models can be used. For example, reaching the error state during simulation can be represented as a search/optimization problem, so Search Based Testing (SBT)[16] can be used. From the models we can automatically generate a *fitness function* to guide the search. Common heuristics such as *approximation level* and *branch distance* of the OCL constraints would be used for the fitness function. Due to size constraints, the investigated testing strategies are reported in [15], where we also proposed a *novel* fitness function that exploits the time properties of the UML/MARTE models.

The use of models for SBT in the case of RTES system testing is essential. In fact, to have effective heuristics (i.e., the fitness function) we need to have precise knowledge of the error states. This information is easily added in the models using stereotypes (Section 3.4). All the relevant states/transitions that lead to those error states can be exploited for the automatic derivation of the fitness function. On the other hand, if we have a simulator but no model, it is unlikely that it would be possible to automatically reverse-engineer all this necessary information from the code alone. Therefore, the fitness function would be necessarily written by hand, with all the related downsides that this choice brings.

In some relevant cases [15], it is possible to automatically derive very precise fitness functions. This happens when time constraints need to be satisfied (a typical case in RTES), e.g., a signal should be received within 10 milliseconds. A test case for which that signal is received after nine milliseconds gives more information than a test case in which the same signal is immediately received after one millisecond (notice that in both cases the constraint is satisfied). SBT can automatically exploit this information by focusing the search on simulator configurations that are more likely to yield a deadline miss. A tester

does not need to write these heuristics, they are in fact automatically derived from the environment models. This is essential, because in general software testers do not have the expertise to write proper fitness functions for search algorithms.

The results in [15] show that our modeling methodology can be used for a fully automated system testing that is effective in revealing faults in industrial RTES. Although different testing strategies can be designed (e.g., Random Testing and SBT), the environment modeling methodology described here would still remain the same.

5. Case Studies

To evaluate the proposed methodology for environment modeling, we applied it on two industrial RTES. The application domains of the systems were entirely different. Because we cannot provide full details of the systems due to confidentiality restrictions, we are providing only a brief description. One of the RTES case studies (Case A) was a sorting system, which was part of an automated bottle recycling machine (developed by Tomra). The system communicated with a number of sensors and actuators to guide recycled items through the recycling machine to their appropriate destinations. The second RTES was a marine seismic acquisition system (Case B). One of the responsibilities of that system was to control the movement of seismic cables, where each cable had a large number of sensors and seismic vibrators, among other equipments. The system regularly communicated with these components and was responsible for managing the life cycle and connections for these components (among other things). We provide a summary of the environment models developed for both the case studies in Table 1.

For Case A, the RTES was configurable as three different types of systems; therefore the number of environment components was large. But most of the components' behavior could be modeled with a couple of states. The highest number of states was 18. Many components inherited a parent component behavior, i.e., its state machine. That was the case for example for DiskMotor and Motor in Fig. 1.

Though the number of components for Case B was more limited than for Case A, the number of instances for some of the components in the environment was very large (e.g., thousands of sensors of the same type communicating with the SUT), thus leading to many instances of executing state machines during simulation. The complexity of component state machines was also on average much higher than for Case A.

One important conclusion is that, in both cases, we were able to model the RTES environments with the subset of UML and MARTE that we identified and the lightweight extensions that we proposed. The models were sufficient to generate simulators that could be used to support large-scale test automation. In one of our industrial case study, using random testing and the SBST strategy described above, combined with using the environment model to identify error states (oracle), new critical faults were detected.

For both case studies, the number of components identified at the time of domain modeling was larger than what was finally required. During successive revisions and based on insight obtained through behavioral modeling, some components turned out to be unnecessary and were removed from the domain model. One practical challenge is that it was not easy in practice to identify the right level of abstraction to model the behavior of environment components. Sub-machines were widely used to incrementally refine the behavioral models until the right level of detail was achieved to simulate the behavior of component from the viewpoint of the SUT.

Table 1. Summary of the environment models of the two industrial RTES.

Industry Case	# of env. components	Stateful components	Average # of states	Max states in a component	Max transitions in a component
Case A	55	43	~3	18	40
Case B	5	4	~12	19	29

6. Conclusion

In this paper, we have discussed a methodology for modeling the environment of a Real-Time Embedded System (RTES) in order to enable black-box, system test automation, which is usually performed by test engineers who are not informed of the design specifics of the RTES. For practical reasons and to facilitate its adoption, the methodology is based on standards: UML, MARTE profile, and OCL for modeling the structure, behavior, and constraints of the environment. We, and this is part of our methodology, made a conscious effort to minimize the notation subset used from these standards. We briefly discussed how the environment models are used to generate automated system test cases and a simulator of the environment to enable testing on the development platform. One advantage is that the methodology also allows more focus on the testing for critical and hazardous conditions in the RTES environment as environment failures and possible error states due to faults in the RTES implementation are explicitly modeled.

We modeled the environment of two industrial RTEs in order to investigate whether our methodology and the notation subsets selected were sufficient to fully address the need for automated system testing. Our experience showed that was the case. In particular, by using our environment models to derive test cases and oracles, it was possible to automatically find new, critical faults in one of the industrial case studies using fully automated, large scale random and search-based testing.

Acknowledgements

The work presented in this paper was supported by Norwegian Research Council and was produced as part of the ITEA 2 VERDE project. We are thankful to Christine Husa, Tor Sjøwall, John Roger Johansen, Erling Marhussen, Dag Kristensen, and Anders Emil Olsen, all from Tomra, for their crucial support.

7. References

- [1] OMG, "Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0, <http://www.omg.org/spec/MARTE/1.0/>," ed, 2009.
- [2] T. Kishi and N. Noda, "Aspect-oriented Context Modeling for Embedded Systems," presented at the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 2004.
- [3] G. Karsai, S. Neema, and D. Sharp, "Model-driven architecture for embedded software: A synopsis and an example," *Science of Computer Programming*, vol. 73, pp. 26-38, 2008.
- [4] K. S. Choi, S. C. Jung, H. J. Kim, D. H. Bae, and D. H. Lee, "UML-based Modeling and Simulation Method for Mission-Critical Real-Time Embedded System Development," presented at the IASTED International Conference Proceedings, 2006.
- [5] C. Kreiner, C. Steger, and R. Weiss, "Improvement of Control Software for Automatic Logistic Systems Using Executable Environment Models," presented at the EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO, 1998.
- [6] J. Axelsson, "Unified Modeling of Real-Time Control Systems and Their Physical Environments Using UML," presented at the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01), 2001.
- [7] H. Gomaa, *Designing Concurrent, Distributed And Real-Time Applications With UML*: Addison-Wesley Educational Publishers Inc, 2000.
- [8] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*: Elsevier, 2008.
- [9] M. Auguston, M. J. B, and M. Shing, "Environment behavior models for automation of testing and assessment of system safety," *Information and Software Technology*, vol. 48, pp. 971-980, 2006.

- [10] L. Du Bousquet, F. Ouabdesselam, J. L. Richier, and N. Zuanon, "Lutess: a specification-driven testing environment for synchronous software," presented at the ICSE '99: Proceedings of the 21st International Conference on Software Engineering, Los Angeles, California, United States, 1999.
- [11] K. G. Larsen, M. Mikucionis, and B. Nielsen, "Online Testing of Real-time Systems Using Uppaal," in *Formal Approaches to Software Testing*. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005.
- [12] M. Heisel, D. Hatebur, T. Santen, and D. Seifert, "Testing Against Requirements Using UML Environment Models," in *Fachgruppentreffen Requirements Engineering und Test, Analyse & Verifikation*, 2008, pp. 28-31.
- [13] N. Adjir, P. Saqui-Sannes, and K. M. Rahmouni, "Testing Real-Time Systems Using TINA," in *Testing of Software and Communication Systems*. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009.
- [14] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*: Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.
- [15] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing," *Technical Report, Simula Research Laboratory*, 2010.
- [16] P. McMinn, "Search-based Software Test Data Generation: A Survey," *Software Testing Verification and Reliability*, vol. 14, pp. 105-156, 2004.

Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing

Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand

In: Proceedings of the 22nd IFIP International Conference on Testing Software and Systems. Springer Berlin / Heidelberg, 2010, pp. 95-110

Abstract—Testing real-time embedded systems (RTES) is in many ways challenging. Thousands of test cases can be potentially executed on an industrial RTES. Given the magnitude of testing at the system level, only a fully automated approach can really scale up to test industrial RTES. In this paper we take a black-box approach and model the RTES environment using the UML/- MARTE international standard. Our main motivation is to provide a more practical approach to the model-based testing of RTES by allowing system testers, who are often not familiar with the system design but know the application domain well-enough, to model the environment to enable test automation. Environment models can support the automation of three tasks: the code generation of an environment simulator, the selection of test cases, and the evaluation of their expected results (oracles). In this paper, we focus on the second task (test case selection) and investigate three test automation strategies using inputs from UML/MARTE environment models: Random Testing (baseline), Adaptive Random Testing, and Search-Based Testing (using Genetic Algorithms). Based on one industrial case study and three artificial systems, we show how, in general, no technique is better than the others. Which test selection technique to use is determined by the failure rate (testing stage) and the execution time of test cases. Finally, we propose a practical process to combine the use of all three test strategies.

1. Introduction

Real-time embedded systems (RTES) represent a major proportion of the software being developed [1]. The verification of their correctness is of paramount importance, particularly when these RTES are used for business or safety critical applications (e.g., controllers of nuclear reactors and flying systems). Testing RTES is particularly challenging since they operate in a physical environment composed of possibly large numbers of sensors and actuators. The interactions with the environment can be bound by

time constraints. For example, if the RTES of a gate is informed by a sensor that a train is approaching, then the RTES should command the gate to close down before the train reaches the gate. Missing such time deadlines can have disastrous consequences in the environment in which the RTES works. In general, the timing of interactions with the real-world environment in which the RTES operates can have a significant effect on the resulting behavior of test cases.

In this paper our objective is to enable the black-box, automated testing of RTES based on environment models. More precisely, our goal is to make such environment modeling as easy as possible, and allow the testers to automate testing without any knowledge about the design of the RTES. This is typically a practical requirement for independent system test teams in industrial settings. In addition, the test must be automated in such a way to be adaptable and scalable to the specific complexity of a RTES and available testing resources. By adaptable, we mean that a test strategy should enable the test manager to adjust the amount of testing to available resources, while retaining as high a fault revealing power as possible.

The system testing of a RTES requires interactions with the actual environment or, when necessary and possible, a simulator. Unfortunately, testing the RTES in the real environment usually entails a very high cost and in some cases the consequences of failures would not be acceptable, for example when leading to serious equipment damage or safety concerns. In our context, a test case is a sequence of stimuli, generated by the environment or its simulator, that is sent to the RTES. If a user interacts with the RTES, then the user would be considered as part of the environment as well. There is usually a great number and variety of stimuli with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. A test case can also contain changes of state in the environment that can affect the RTES behavior. For example, with a certain probability, some hardware components might break, and that has effect on the expected and actual behavior of the RTES. A test case can contain information regarding when and in which order to trigger such changes.

Testing all possible sequences of environment stimuli/state changes is not feasible. In practice, a single test case of an industrial RTES could last several seconds/minutes, executing thousands of lines of code, generating hundreds of threads/processes running concurrently, communicating through TCP sockets and/or OS signals, and accessing the

file system for I/O operations. Hence, systematic testing strategies that have high fault revealing power must be devised.

The complexity of modern RTES makes the use of systematic testing techniques, whether based on the coverage of code or models, difficult to apply without generating far too many test cases. Alternatively, manually selecting and writing appropriate test cases based on human expertise for such complex systems would be far too challenging and time consuming. If any part of the specification of the RTES changes during its development, a very common occurrence in practice, then many test cases might become obsolete and their expected output would potentially need to be recalculated manually. The use of an automated oracle is hence another essential requirement when dealing with complex industrial RTES.

In this paper we present a Model-Based Testing (MBT) [2] methodology to carry out system testing of RTES in a fully automated, adaptable, and scalable way. We tailor the principles of Adaptive Random Testing (ART) [3] and Search-Based Testing (SBT) [4] to our specific problem and context. For our empirical evaluation, we use Random Testing (RT) [5] as baseline. One main advantage of ART and SBT is that it can be tailored to whatever time and resources are available for testing: when resources are expended and time is up, we can simply stop their application without any side effect. A coverage-based strategy could not be, for example, interrupted at any time. Furthermore, ART and SBT attempt, through different heuristics, to maximize the chances to trigger a failure within time constraints. We will also see how their combined use can be helpful to gain the most out of testing resources in practice. The RTES under test (SUT) is treated as a black box: no internal detail or model of its behavior is required, as per our objectives. The first step is to model the environment using the UML standard and its MARTE profile, the latter being necessary to capture real-time properties. The use of international standards rather than academic notations is dictated by the fact that our solutions are meant to be applied by our industry partners. Environment models support test automation in three different ways:

- The environment models describe some of the structural and behavioral properties of the environment. Given an appropriate level of detail, they enable the automatic generation of an environment simulator to satisfy the specific needs of software testing.

- The models can be used to generate automated oracles. These could for example be invariants and error states that should never be reached by the environment during the execution of a test case (e.g., an open gate while a train is passing). In general, error states can model unsafe, undesirable, or illegal states in the environment. We used error states as oracles in our case studies.
- Test cases can be automatically selected based on the models, using various heuristics to maximize chances of fault detection. In our case studies we use ART and SBT.

In this paper we focus on the third item above and assess RT, ART, and SBT on the production code of a real industrial RTES. Due to space constraints, and because our focus in this paper is test automation, we do not explain in detail how to use UML/- MARTE to model the environment of a RTES and how simulator code can be automatically generated (which we investigated in [6]). To the best of our knowledge, no MBT automation results for ART and SBT on an actual RTES have ever been reported in the research literature. Since no freely available RTES was available, we also constructed three different artificial RTES in order to extend our investigation and better understand the influence of various factors on test cost-effectiveness such as the failure detection rate. The use of publicly available artificial RTES will also facilitate future empirical comparisons with our work since, due to confidentiality constraints, our industrial case study cannot be made public.

The paper is organized as follows. Section 2 provides an overview of related work. How the context is modeled and simulated is shortly discussed in Section 3. Section 4 describes the different strategies we used to generate test cases. Their empirical validation is described in Section 5 and threats to validity are discussed in Section 6. Finally, Section 7 concludes the paper.

2. Related Work

A large body of literature has been dedicated to test RTES. For reason of space, here we can only give a very brief and incomplete overview.

Most of the approaches to test RTES are based on violating their timing constraints [7] or checking their conformance to a specification [8]. The specification is generally a formal model of the system and this model is then used to generate test cases. A number of approaches have been proposed over the years to address the above problem. The most

widely discussed approaches model the system using Timed Automata [9]. A number of Timed Automata extensions, such as Timed I/O Automata [10], have also been used for conformance testing. For the same purpose, UML statechart [11], Extended Finite State Machines [12] and Attributed Event Grammar [13] have also been used.

There are several works using SBT techniques for testing different aspects of RTES [14], as for example identify deadline misses [15] and testing functional properties [16].

The work presented here is significantly different from most the above approaches as we adopt, for practical reasons presented above, a black-box approach to system testing that relies on modeling the RTES environment rather than its internal design properties. As noted above, this is of practical importance as independent system test teams usually do not have easy access to precise design information. Most existing work does not focus on system testing, hence their emphasis on modeling the RTES internal behavior and structure. Another difference of practical importance, though this is not detailed in this paper, is that we use UML and its standard extensions for modeling the environment. Last but not least, as opposed to published case studies (e.g., [13, 12]), we assess our test strategies on the actual production code of an industrial RTES.

3. Environment Modeling and Simulation

For RTES system testing, software engineers would typically be responsible for developing the environment models. Therefore, the modeling language should be familiar to them and therefore based on software engineering standards. In other words, it is important to use a modeling language for environment modeling that is widely accepted and used by software engineers. Furthermore, standard modeling languages are widely supported in terms of tools and training. The Unified Modeling Language (UML) and its extensions are therefore a natural choice to consider in our context. Several modeling and simulation languages are available and can be used

Several modeling and simulation languages are available and can be used for modeling and simulating the context (e.g., DEVS [17]). But in our case using these simulation languages raises a number of issues, including the fact that software engineers in the development team are usually not familiar with the notations and concepts of such languages.

Higher level programming languages (such as Java or C) can also be used as simulation languages. The major problem with the use of such languages is the low level of abstraction at which they “model” the environment. The software engineers will have to deal with all the programming language constructs (such as threads) while at the same time trying to focus on the details of the environment itself.

RTES testing through an environment simulator faces the question of how time is handled. Indeed, many properties of the RTES depend on whether some time constraints are fulfilled or not. Ideally, we would like to be able to simulate the passing of time in a deterministic way, but it is not always possible for large and complex RTES.

The opposite approach to time simulation would be to run the RTES with its simulated environment using the real clock of the CPU used to run the empirical analysis. On one hand, it has the benefit that we do not have any particular constraint on the type of RTES that can be analyzed. On the other hand, it adds noise and variance in the scheduled time events. If time constraints of the RTES are very tight (e.g., in the order of few milliseconds), then this approach is not a viable option.

In our work, we have used UML/MARTE as a simulation language. Models are developed in UML as classes and their state-machines. These models are then transformed into Java using model to text transformations. The activities and actions are written in Java and are converted into Java method calls. This was appropriate for the RTES considered in this paper. For other types of RTES, different programming languages could be necessary. Notice that our methodology is general. We chose Java only for practical reasons. In particular, in our empirical analyses we did not face the problem of the garbage collector interfering with time properties. The garbage collector was never called during the execution of a test case.

4. Automated Testing

4.1. Test Case Representation

In our context, a test case execution is akin to executing the environment simulator. Each state machine represents a component of the environment. There can be more instances of a state machine with different settings to represent different sensors/actuators of the same type. For example, in a gate controller RTES, we can have a state machine representing the

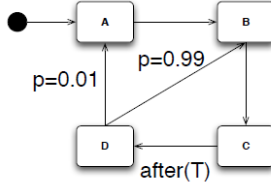


Figure 1. Example of a reduced UML/MARTE state machine

trains. For each simulated train we will have an independent running instance of that state machine. The domain model is used to identify how many instances can or should run in parallel for each state machine. Based on the domain model, there could be different possible configurations of the environment, but in this paper we focus only on one fixed configuration.

In the behavioral models of the environment (i.e., the state machines) there can be non-deterministic parts. For example, a timeout transition could be triggered within a minimum and a maximum time value but the exact value cannot be determined. This is very typical when real-world components are modeled, in which for example there is always a natural variance when time-related properties are represented. Another example is when we assign probabilities p in the models to represent failure scenarios, as for example the breakdown of sensors/actuators. In our context, input data of a test case are the choice of the actual values to use in these non-deterministic events.

In our modeling methodology, we have non-deterministic choices only in the transitions between states. They can be in the trigger, the guard and the action of the transition. A transition might be taken several times, and this number might be unknown before executing the test case. Therefore, for each instance of the environment state machines, for each non-deterministic choice, we allocate in the test case a vector of possible values. The length of this vector is l . Each time such non-deterministic choice needs to be made, a value from the corresponding vector is selected. Because the vector has finite length l , it is used as a ring: The values are taken in order, and after l request for values, it starts again from the beginning of the vector. Figure 1 shows an example.

Let the transition $C \rightarrow D$ have a non-deterministic choice in $[0,1]$, for example the timeout $T \in [0,1]$. Given for example $l = 2$, we would have a data vector containing for example $\{0.4, 0.32\}$. The first time the transition $C \rightarrow D$ is taken, the value 0.4 is used for

the non-deterministic choice. The second time, the value 0.32 is used. The third time, the value 0.4 is used again, and so on.

Given n state machine instances, and m non-deterministic choices in each of them (for simplicity, because in general instances of different machines will have a different number of non-deterministic choices), we would have that each test case contains $L = n * m * l$ values, which can be represented as a vector. The choice of l is arbitrary but has significant consequences. On one hand, a small number of possible values could make it impossible to represent sequences of event patterns that lead to failures in the RTES. On the other hand, a high number of possible values will lead to long vectors and might harm the effectiveness of test selections techniques such as ART and SBT (discussed in more details in the next sections).

In our case studies, the values to include in the test case data are chosen before the execution of the test cases. This means that the domain of these values should be static and not depending on the dynamic execution of the test cases. For example, if a variable is constrained within a minimum and maximum limit, then these boundaries should be known before test execution. This is the case for the industrial RTES analyzed in this paper and for other RTES we have worked with. When this is not the case, we would need to enable the choice of non-deterministic options at runtime.

4.2. Testing Strategies

As described in the previous section, a test case can be seen as a vector V . Elements in this vector can be of different types, but their domain of valid values should be known. Given $D(i)$ the domain of the i^{th} variable in V , we obtain that the number of possible valid test cases is $\prod |D(i)|$, which is an extremely large number. An exhaustive execution of all possible test cases is infeasible.

In this paper we consider the testing problem of sampling test cases to detect failures of the RTES with automated oracles derived from the environment models. For all test strategies, the oracle checks whether a transition to an error state specified in the model occurs during test execution. We choose and execute test cases one at a time. We stop sampling test cases as soon as a failure has been found. A test strategy that requires the sampling of fewer test cases to detect failures should obviously be preferred.

The simplest, automated technique to choose test cases is Random Testing (RT). For each variable in V , we simply sample a value from its domain with uniform probability. Although RT can be considered to be a naive technique, it has been shown to be effective in many testing situations [18, 19].

Another technique that we investigate is Adaptive RT (ART) [3], which has been proposed as an extension of RT. The underlying idea of ART is that diversity among test cases should be rewarded, because failing test cases tend to be clustered in contiguous regions of the input domain. ART can be automated if one can define a meaningful similarity function for test cases. To the best of our knowledge, we are aware of no previous application of ART to test RTES. In this paper we use the basic ART algorithm described in [3].

Because in our case studies all the variables in V are numerical, for the distance between two test case data vectors $V1$ and $V2$ we use the following $dis(V1, V2) = \sum_i |abs(V1(i) - V2(i))|/|D(i)|$. We sum the absolute difference of each variable weighted by the cardinality of the domain of that variable. Often, these variables represent the time in timeout transitions. Therefore, ART rewards diversity in the triggering time of events.

In this paper we also investigate the use of search algorithms to tackle the testing of RTES. In particular we consider the use of Genetic Algorithms (GAs), which are the most used search algorithms in the literature on search-based testing (SBT) [14]. To use search algorithms to tackle a specific problem, a fitness function needs to be defined tailored to solve that problem. Search algorithms exploit the fitness function to guide the search toward promising areas of the search space. The fitness function is used to heuristically evaluate how “good” a test case is. In our case, the fitness function is used to estimate how close a test case is from triggering a failure in the RTES, that is when at least one component of the environment enters an error state. This is once again determined by analyzing the environment models.

To tackle the testing problem described in this paper, we developed a novel fitness function f that can be seen as an extension of the fitness functions that are commonly used for structural testing [4] and MBT [20]. In our case, the goal is to minimize the fitness function f . If at least one error state is reached when a test case with test data V is executed, then $f(V) = 0$. For each error state E in each state machine instance we employ the so called approach level A and branch distance B . The approach level calculates the minimum

number of transitions in the state machine to reach an error state from the closest executed state. The branch distance is used to heuristically score the evaluation of the Object Constraint Language (OCL) constraints in the closest executed state from which the approach level is calculated. The branch distance is used to guide the search to find test data that satisfy those OCL constraints. A transition could be triggered several times but never executed because the guard fails. For the branch distance, we calculate it every time but then we only consider the minimum value it obtains. Because the branch distance is less important than the approach level, it is normalized in the range [0,1]. We use the following normalizing function: $nor(x) = x/(x+1)$, which has been shown to be better than other normalizing functions used in the literature [21]. Notice that, in the case of MBT, it is not always possible to calculate the branch distance when the related transition has never been triggered. In these cases, we assign to the branch distance B its highest possible value.

The extension of the fitness function we make in this paper exploits the time properties of the RTES. Some of the transitions are triggered when a time-threshold is violated. For example, an error state could be reached if a sensor/actuator does not receive a message from RTES within a time limit. If such transitions exist on the path toward the execution of the error states, then we need a way to reward test data that get the execution closer to violate those time constraints. If a transition is taken after a threshold z , then we calculate the maximum consecutive time t the state machine stays in the state from which that transition can be triggered (this would be the same state from which the approach level is calculated from). Then, to guide the search we can use the following heuristic $T = z - t$, where $t \leq z$.

Finally, the fitness function f for a test data vector V is defined as:

$$f(V) = \min_E A_E(V) + nor(T_E(V)) + nor(B_E(V))$$

Notice that, to collect information such as the approach level, the source code of the simulator needs to be instrumented. This is automatically done when this code is generated from the environment models.

Once the fitness function is defined, we can use it to guide the GA to select test cases. But GAs have many parameters that need to be set. In this paper we use a Steady State GA [4]. We employ rank selection with bias 1.5 to choose the parents. A single point crossover is employed with probability $P_{\text{crossover}} = 0.75$. This operator chooses a random point inside the data vectors V of the parents s_x and s_y . The elements in the data vector after that splitting

Table 1. Summary of the state machines of the environment of the industrial RTES. NDC stands for “Non-Deterministic Choice”.

State Machine	States	Transitions	Error States	Instances	NDCs for Instance
S1	19	29	1	10	6
S2	4	7	0	11	2
S3	3	8	1	1	0
S4	5	5	0	1	0

point are swapped between the two parent solutions. Each of the L elements in a data vector is mutated with probability $1/L$. A mutation consists of replacing a value with another one at random from the same domain. The population size is chosen to be 10. The optimal configuration of search algorithms is in general problem dependent [22]. Due to the large computational cost of running our empirical analysis, we have not tuned the GA. We simply use reasonable parameter values given in the literature of GAs.

5. Empirical Study

5.1. Case Study

To validate the novel approach presented in this paper, we have applied it to test an industrial RTES. The analyzed system is a very large and complex controller that interacts with several sensors/actuators. The company that provided the system is a market leader in its field. For confidentiality reasons we cannot provide full details of the system. Information of the environment models of this RTES is provided in Table 1. Notice that for this case study there are several state machines, and for each of them there can be one or more instances running in parallel at the same time. For each test case, 23 instances of state machines run in parallel, each of them can start several threads. The total number of non-deterministic choices (NDCs) is 82. The UML/MARTE context models were developed in IBM Rational Software Architect. Constraints, such as guards, were expressed in OCL.

To facilitate future comparisons with the techniques described in this paper, it would be necessary to also employ a set of benchmark systems that are freely available to researchers. Unfortunately, we have not found any RTES satisfying this criterion. Therefore, in addition to our industrial case study, we have designed three artificial RTES, called AP1, AP2 and AP3. Two of them are inspired by the industrial RTES used in this paper, whereas the third is inspired by the control gate system described in [12]. The RTES are written in Java to facilitate their use on different machines and operating systems. For

the same reason, the communications between the RTES and their environments are carried out through TCP. The use of TCP was also essential to simplify the connection of the RTES with its environment. For example, if the simulator of the environment is generated from the models using a different target language (e.g., C/C++), then it will not be too difficult to connect to the artificial RTES written in Java. These RTES are all multithreaded. Table 2 summarizes the properties of these artificial RTES. In each of them, there is only one error state. We introduced by hand a single non-trivial fault in each of these RTES.

5.2. Experiments

We have carried out two different sets of experiments. One for the artificial problems, and one for the industrial RTES. In all these experiments, the value l for the nondeterministic choices is set to $l = 3$. This means that the number of input variables in each test case is 60 for AP1, 12 for AP2, 54 for AP3 and finally 246 for the industrial RTES.

In the first step of the experiments, we ran RT, ART and GA on each of the three artificial problems. Because the execution of a single test case takes 10 seconds, we stop each algorithm after 1000 sampled test case or as soon as one of the error state is reached. Notice that the value 10 seconds is fixed, and it does not depend on the used execution platform. Using faster hardware would not change the amount of time required to run these experiments. The only requirement is that the hardware used for the experiments is fast enough to sustain the CPU load without introducing delays higher than a few milliseconds. Because in these simulations most of the time the CPU is in idle state, the computers used in the experiments were appropriate.

For each test strategy and each case study, we ran the algorithms 100 times with different random seeds. Because these algorithms are randomized, a large number of experiments is required to obtain statistically significant results. The total number of sampled test cases is hence at most $3 * 3 * 1000 * 100 = 900,000$, which can take up to 104 days on a single computer. To cope with this problem, we used a cluster to run these

Table 2. Properties of the three artificial problems. LoC stands for “Lines of Code”, whereas NDC stands for “Non-Deterministic Choice”.

Artificial Problem	LoC of RTES	LoC of Environment	State Machines	States	Transitions	Instances	Total NDCs
AP1	227	259	1	5	7	10	20
AP2	409	271	1	5	7	2	4
AP3	337	318	2	9	13	5	18

Table 3. Success rate (out of 100 runs) for the three artificial problems.

Algorithm	AP1	AP2	AP3
RT	6	35	49
ART	0	40	74
GA	90	21	31

experiments.

Given an upper bound of 1000 test cases, it is not always the case that any of the test strategies is able to trigger a failure in the RTES. In Table 3 we report how many times each algorithm was able to do so out of the 100 experiments. Because the process of detecting failures in 100 experiments can be considered to be a binomial process with unknown probability [23], we use the Fisher Exact test to compare the success rate of RT with the ones of ART and GA. The significance level of the tests is set to 0.05. Results show that the only case in which there is no significant difference in the success rate is for problem AP2 when RT is compared to ART.

The second set of experiments has been carried out on an industrial RTES. In system testing of RTES, the simulation of the environment can in general be run for any arbitrary amount of time. But there should be enough time to render possible the execution of all the functionalities of the RTES. For example, in the RTES for a train/gate controller, we should run the simulation at least long enough to make it possible for a train to arrive and then leave the gate. Choosing for how long to run a simulation (i.e., a test case) is conceptually the same as the choice of test sequence length in unit testing [24] (i.e., many short test cases or only few ones that are long?). But in contrast to unit testing in which often the execution time of a test case is in the order of milliseconds, in the system testing of RTES we have to deal with much longer execution time. In this paper, we run each test case for 20 seconds. This choice has been made based on the properties of the RTES and discussions with its software testers.

We evaluated the use of RT, ART and GA to find failures in this RTES. We could not run this empirical analysis on a cluster due to technical reasons. We used a single dedicated computer, and it took nearly ten days to run these experiments. The failure rate of the SUT

Table 4. Success rate (out of 100 runs) for the three artificial problems.

Algorithm	Min	Median	Mean	Max	SD
RT	1	73.0	131.9	912	164.9
ART	1	75.5	104.6	525	99.7
GA	1	99.0	160.0	767	155.2

Table 5. Results of the statistical tests for the data in Table 4

Comparison	<i>t</i> -tests p-value	Cohen D	U-test p-value	Vargha-Delaney A
RT vs ART	0.1588	0.2012	0.9708	0.5015
RT vs GA	0.2150	-0.1768	0.0334	0.4129
ART vs GA	0.0030	-0.4272	0.0193	0.4042

in these experiments was quite high, so we did not use any upper bound for the number of sampled test cases. The results of experiments are shown in Table 4.

To analyze the results in a sound manner we carried out a set of statistical tests on the data presented in Table 4. We used parametric *t*-tests to see whether there is any statistical difference between the mean values of sampled test cases among the three analyzed algorithms. The scientific or practical significance of these differences is evaluated using the Cohen D coefficient. We also carried out non-parametric Mann-Whitney U tests to see whether any of the results of these algorithms is stochastically greater than the others. The scientific significance of this test is measured with the Vargha-Delaney A statistic. For both *t*-tests and Mann-Whitney U tests the significant level is set to 0.05. For the Cohen D coefficient (value *d*), we classify the effect size as follows [25]: small for $abs(d) = 0.2$, medium for $abs(d) = 0.5$, and finally large for value $abs(d) = 0.8$. In the case of Vargha-Delaney A statistic (value *a*), we use the following classification [26]: small for $abs(a - 0.5) = 0.06$, medium for $abs(a - 0.5) = 0.14$ and large for $abs(a - 0.5) = 0.21$. Table 5 summarizes the results of these statistical tests.

5.3. Discussion

In the results of the experiments on the artificial problems shown in Table 3, we can see that no testing technique generally dominates the others. GA is statistically better on the first problem, but it is the worst on the other two problems. Regarding RT and ART, they are equivalent on the second problem, but RT is best on the first, whereas ART is best on the third problem.

The results in Table 3 for GA can be precisely explained. Covering all the nonerror states and transitions in the environment models of these problems is very easy, practically all test strategies achieve this. The only difficult part is the transition to the error state. For the first problem AP1, that transition is a time transition with no guard. After a time threshold, that transition is triggered. The novel fitness function proposed in this paper can take advantage of this information, rewarding test cases that get closer to violate that time

constraint. In fact, for each test case we can automatically calculate the time that it spends in the state that could lead to the error state. This automated fitness function produces an easy fitness landscape that can be efficiently searched by GA. This explains the fact that GA gets to the error state 90% of the time, whereas RT reaches it only in 6% of the time. However, why do we obtain so much worse results in the other two problems AP2 and AP3? The reason is that the fitness function in these cases is practically a needle-in-the-haystack function. In the transition to the error state, there is a guard that is checking whether one Boolean variable is equal to true. The value of this variable depends on the interactions with the SUT, particularly whether a specific message has been received or not. This type of guard in search-based testing is a known, very difficult problem denoted as the flag problem [27]. In this case, the fitness function provides no gradient, and this makes the search difficult. Unfortunately, testability transformations [27] cannot be used in this case, because in our context the SUT is a black box. Even if we had access to the SUT, it would still be problematic, because we are aware of no work dealing with the flag problem for the system testing of concurrent programs. Though the above issue is a limitation, in practice, we can automatically determine before running GA whether it will work.

Though we can explain why GA does not work well on AP2 and AP3, why does it behave even worse than RT? The reason is exactly the same for which ART is better than RT: the diversity of the test cases. If there is no gradient in the fitness function, all the sampled test cases would have same fitness value (i.e., the fitness landscape would have a large plateau). So any new sampled test case would be accepted and added to the next generation in GA. The crossover operator does not produce any new value in the data vector V , it simply swaps values between two parent test cases. The mutator operator does only small changes to a data vector, because on average only one variable is mutated. During the search, the offspring have genetic material (i.e., the data vectors) that is similar to the one of the parents. Therefore, the diversity of test cases during GA evolution is much lower than the one of RT. If the hypothesis of contiguous regions of faulty test cases is true for a RTES, then, when there is no gradient in the fitness function, we would a-priori expect this following relationship regarding the performance of testing strategies: $GA \leq RT \leq ART$. For problems AP2 and AP3, this is verified in the results of Table 3.

In the experiments on the industrial RTES, we can see that GA is statistically worse than the other approaches, although the difference is only small/medium in size from a scientific point of view. The results on the industrial RTES shown in Table 4 are important to stress out that the choice of a testing strategy is also heavily dependent on when the SUT is tested. The version of the industrial RTES used in this paper was not a finished product. It was in an early phase of development. The types of failure scenarios introduced with our models were not something that was fully tested before. This explains the high failure rate shown in Table 4. Notice that the failure rate θ can be simply estimated from the mean value of RT, i.e. $\theta = 1/\text{mean}(RT)$. The reason is that RT follows a geometric distribution with parameter θ , therefore $\text{mean}(RT) = 1/\theta$. In our case, we have $\theta = 1/131.9 = 0.007$, which can be considered to be a high failure rate.

5.4. Practical Guidelines

For high failure rates, it makes sense to use a simple RT instead of more sophisticated techniques, since the expected number of sampled test cases would be low on average. In practice, we would expect high failure rates at the beginning of the testing phase. The failure rate would hence be expected to decrease throughout the development process as faults get fixed. Therefore, we would expect to get good results for RT at the beginning, but then more sophisticated techniques could be required at later stages.

Our results lead us to suggest the following heuristics to apply RT, ART, and SBT in practice: In the early stages of development and testing, when failure rates are still high, one should use RT as it will be very efficient and quick to detect the first failure, without requiring any overhead like ART or SBT. One exception to this rule is when the time of executing a test case is high (e.g., in the order of several seconds or minutes), where we then suggest to use ART as one should enforce test execution diversity to prevent the execution of too many test cases. Once the failure rate decreases due to the fixing of *easy-to-detect* faults, then use SBT, but only if a proper fitness function can be derived automatically from the models, that is a fitness function that is likely to provide effective guidance for the search of failing test cases. Otherwise, use RT. ART should not be used when the failure rate is low as the overhead of distance calculations would get too high, due to the large number of test cases executed.

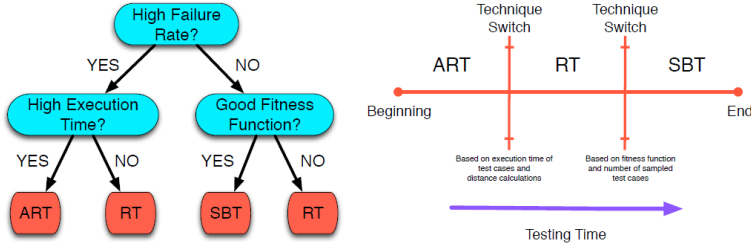


Figure 2. Decision tree and application timeline of the three analyzed testing strategies.

Figure 2 summarizes the above heuristic in a decision tree and it shows when to apply each testing technique. We provide practical advice regarding when to switch from ART to RT below. But for the switch from RT to SBT, we need more empirical/theoretical analyses to provide practical guidelines.

In the literature, it has been shown that ART can be twice as fast as RT [3]. Let us consider t_{tc} the execution time of a test case, t_{dis} the execution time of a distance calculation with d the total number of distances computed, θ the failure rate, $E[RT]$ and $E[ART]$ the expected number of test cases sampled by RT and ART. We know that $E[RT] = 1/\theta$ and that, under optimal conditions, $E[ART] = E[RT]/2$. We can develop a heuristic that is based on the following equation: $E[RT] \cdot t_{tc} = E[ART] \cdot t_{tc} + d \cdot t_{dis}$, which is a *loose approximation* to determine the failure rate θ above which ART is going to yield better results than RT. From that equation, it follows $\theta^* \approx t_{dis}/4 \cdot t_{tc}$. This optimal threshold for ART for the failure rate can be estimated before test execution. Finally, we can suggest to run ART for $\frac{1}{2} \theta^*$ iterations, but only as long as the number of sampled test cases is not high enough to make the decision to switch to SBT. The above recommendations are heuristics and will need to be evaluated and refined as we gather more empirical data.

6. Threats to validity

Due to the complexity of the industrial RTES used in the empirical study of this paper, we could not run the RTES and its simulated environment in such a way to obtain a precise and deterministic handling of clock time. We used the CPU clock instead. This could be unreliable if time constraints in the RTES are very tight, as for example in the order of milliseconds, because these constraints could be violated due to unpredictable changes of load balance in the CPU because of unrelated processes. Although the time constraints in

this paper were in the order of seconds, the problem could still remain. To evaluate whether our results are reliable, we hence selected a set of experiments, and we re-ran them again with exactly the same random seeds. We obtained equivalent results. For example, if RT for a particular seed obtained a failing test case after sampling 43 test cases, then, when we ran it again with the same seed, it was still requiring exactly 43 test cases. However, the experiments were not exactly the same. For example, for debugging purposes we used time stamps on log files. In these time stamps, small variances of a few milliseconds were present, but this did not have any effect on the testing results. Notice that our novel methodology can obviously be applied also when time clocks are simulated.

7. Conclusion

In this paper we proposed a black-box system testing methodology, based on environment modeling and various heuristics for test case generation. The focus on black-box testing is due to the fact that system test teams are often independent from the development team and do not have (easy) access to system design expertise. Our objective is to achieve full system test automation that scales up to large industrial RTES and can be easily adjusted to resource constraints. The environment models are used for code generation of the environment simulator, selecting test cases, and the generation of corresponding oracles. The only incurred cost by human testers is the development of the environment models. This paper, due to space constraints, has focused on the testing heuristics and an empirical study to determine the conditions under which they are effective, plus guidelines to combine them in practice.

In contrast to most of the work in the literature, the modeling and the experiments were carried out on an industrial RTES in order to achieve maximum realism in our results. However, in order to more precisely understand under which conditions each test heuristic is appropriate and how to combine them, we complemented this industrial study with artificial case studies, that will be made publicly available to foster future empirical analyses and comparisons.

We experimented with different testing heuristics, which have the common property to be easily adjustable to available time and resources: Random Testing (RT), Adaptive Random Testing (ART) and Search-Based Testing using Genetic Algorithms (GAs). All these techniques can be adjusted to project constraints as they can be run as long as time

and access to CPU are available. Though RT was originally used as comparison baseline, it turned out to be the best alternative under certain conditions.

On the artificial problems, in one case GA is the best search algorithm, and the difference is very large. But on the other two cases, GA has the worst results, which are due to poor fitness functions. In one case RT and ART are equivalent, but in the other two, RT is better in one case and worse in the other.

However, on the industrial RTES, results are quite different from the artificial case studies: there is no statistical difference between RT and ART, whereas GA is slightly worse than the others (the effect size is between *small* and *medium*). After investigation, this was found to be due to the RTES high failure rate and a fitness function that offered little guidance to the search due to a Boolean guard condition. To support the claims above, we followed a rigorous experimental method based on five types of statistical analyses.

Based on our results, we have provided practical guidelines to apply the three testing techniques described in this paper, i.e. RT, ART, and GA. In fact, none of them dominates the others in all testing conditions and they must be, in practice, combined to achieve better results. However, more empirical and theoretical studies are needed to develop more precise, practical guidelines.

One current limitation of our testing approach is that the domains of valid values for the non-deterministic test inputs need to be static: they should be known before test case execution. Research will need to be carried out to design novel testing strategies for non-deterministic inputs that can only be determined at runtime.

Acknowledgements

The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the ITEA-2 project called VERDE.

8. References

- [1] Douglass, B.P.: Real-time UML: developing efficient objects for embedded systems. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1997)
- [2] Utting, M., Legeard, B.: Practical model-based testing: a tools approach. Elsevier (2007)

- [3] Chen, T.Y., Kuo, F., Merkela, R.G., Tseb, T.: Adaptive random testing: The art of test case diversity. *Journal of Systems and Software (JSS)* (2010) (in press).
- [4] McMinn, P.: Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14(2) (2004) 105–156
- [5] Myers, G.: *The Art of Software Testing*. Wiley, New York (1979)
- [6] Iqbal, M.Z., Arcuri, A., Briand, L.: Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies. In: *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*. (2010)
- [7] Clarke, D., Lee, I.: Testing real-time constraints in a process algebraic setting. In: *IEEE International Conference on Software Engineering (ICSE)*. (1995) 51–60
- [8] Krichen, M., Tripakis, S.: Conformance testing for real-time systems. *Formal Methods in System Design* 34(3) (2009) 238–304
- [9] Alur, R., Dill, D.L.: A Theory of Timed Automata. *Theoretical Computer Science* 126 (1994) 183–235
- [10] En-Nouary, A.: A scalable method for testing real-time systems. *Software Quality Journal* 16(1) (2008) 3–22
- [11] Miicke, T., Huhn, M.: Generation of optimized testsuites for UML statecharts with time. In: *IFIP international conference on testing of communicating systems*. (2004) 128–143
- [12] Zheng, M., Alagar, V., Ormandjieva, O.: Automated generation of test suites from formal specifications of real-time reactive systems. *Journal of Systems and Software (JSS)* 81(2) (2008) 286–304
- [13] Auguston, M., Michael, J.B., Shing, M.T.: Environment behavior models for automation of testing and assessment of system safety. *Information and Software Technology (IST)* 48(10) (2006) 971–980
- [14] Harman, M., Mansouri, S.A., Zhang, Y.: Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Technical Report TR-09-03, King's College* (2009)
- [15] Garousi, V., Briand, L.C., Labiche, Y.: Traffic-aware stress testing of distributed real-time systems based on uml models using genetic algorithms. *Journal of Systems and Software (JSS)* 81(2) (2008) 161–185
- [16] Lindlar, F., Windisch, A., Wegener, J.: Integrating model-based testing with evolutionary functional testing. In: *International Workshop on Search-Based Software Testing (SBST)*. (2010)
- [17] Zeigler, B.P., Praehofer, H., Kim, T.G.: *Theory of modeling and simulation*. Academic press New York, NY (2000)
- [18] Duran, J.W., Ntafos, S.C.: An evaluation of random testing. *IEEE Transactions on Software Engineering (TSE)* 10(4) (1984) 438–444
- [19] Arcuri, A., Iqbal, M.Z., Briand, L.: Formal analysis of the effectiveness and predictability of random testing. In: *ACM International Symposium on Software Testing and Analysis (ISSTA)*. (2010)
- [20] Lefticaru, R., Ipate, F.: Functional search-based testing from state machines. In: *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. (2010) 525–528
- [21] Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. (2010) 205–214

- [22] Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1(1) (1997) 67–82
- [23] Feller, W.: *An Introduction to Probability Theory and Its Applications*, Vol. 1. 3 edn. Wiley (1968)
- [24] Arcuri, A.: Longer is better: On the role of test sequence length in software testing. In: *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. (2010) 469–478
- [25] Cohen, J.: A power primer. *Psychological bulletin* 112(1) (1992) 155–159
- [26] Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25(2) (2000) 101–132
- [27] Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. *IEEE Transactions on Software Engineering* 30(1) (2004) 3–16

Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software

Muhammad Zohaib Iqbal, Andrea Arcuri, Lionel Briand

In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, 2012 (to appear).

Abstract—System testing of real-time embedded systems (RTES) is a challenging task and only a fully automated testing approach can scale up to the testing requirements of industrial RTES. One such approach, which offers the advantage for testing teams to be black-box, is to use environment models to automatically generate test cases and oracles and an environment simulator to enable earlier and more practical testing. In this paper, we propose novel heuristics for search-based, RTES system testing which are based on these environment models. We evaluate the fault detection effectiveness of two search-based algorithms, i.e., Genetic Algorithms and (1+1) Evolutionary Algorithm, when using these novel heuristics and their combinations. Preliminary experiments on 13 carefully selected, non-trivial artificial problems, show that, under certain conditions, these novel heuristics are effective at bringing the environment into a state exhibiting a system fault. The heuristic combination that showed the best overall performance on the artificial problems was applied on an industrial case study where it showed consistent results.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Experimentation, Verification.

Keywords

Automated model-based testing, real-time embedded systems, search-based software engineering, branch distance.

1. Introduction

Real-time embedded systems (RTES) are part of a vast majority of computing devices available today. They are widely used in critical domains where high system dependability is required. These systems typically work in environments comprising of large numbers of interacting components. The interactions with the environment can be bound by time constraints. For example, if a gate controller RTES on a railroad intersection is informed by a sensor that a train is approaching, then the RTES should command the gate to close before the train reaches it. Missing such time deadlines, or missing them too often for soft real-time systems, can lead to serious failures leading to threats to human life or the environment. There is usually a great number and variety of stimuli from the RTES environment with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. Testing all possible sequences of stimuli is not feasible. Hence, systematic automated testing strategies that have high fault revealing power are essential for effective testing of industry scale RTES. The system testing of a RTES requires interactions with the actual environment. Since, the cost of testing in actual environments tends to be high, environment simulators are typically used for this purpose.

In our earlier work, we proposed an automated system testing approach for RTES software based on environment models [1, 2]. The models are developed according to a specific strategy using the Unified Modeling Language (UML) [3], the Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile [4] and our proposed profile for environment modeling [5]. These models of the environment were used to generate an environment simulator [6], test cases, and obtain test oracle [1, 2]. We applied various testing strategies to generate test cases, including search-based strategies, which turned out not to work very well as even Random Testing (RT) [7] fared better.

In our context, a test case is a sequence of stimuli generated by the environment that is sent to the RTES. If a user interacts with the RTES, then she would be considered part of the environment as well. A test case can also include changes of state in the environment that can affect the RTES behavior. For example, with a certain probability, some hardware components might break, and that affects the expected and actual behavior of the RTES. A test case can contain information regarding when and in which order to trigger such changes. So, at a higher level, a test case in our context can be considered as a setting specifying the occurrence of all

these environment events in the simulator. Explicit “error” states in the models represent states that should never be reached if the RTES is correct. If any of these error states is reached, then it implies a faulty RTES. Error states act as the oracle of the test cases, i.e., a test case is successful in triggering a fault in the RTES if an error state of the environment is reached during testing.

In this paper, we further extend the fitness function proposed in [1] to improve the disappointing results we had obtained with search-based testing. For this purpose, we present four new heuristics that are aimed to exploit potentially useful characteristics of the environment models. We evaluate the fault detection effectiveness of the new heuristics and their combinations by first performing a series of experiments on 13 artificial RTES that we developed based on the specifications of two industrial case studies. For all heuristics, we used two search algorithms: Genetic Algorithms (GA) and (1+1) Evolutionary Algorithms (EA). We also ran RT on the problems as a comparison baseline. We then ran the heuristic combination that on average showed best results for the artificial problems on an industrial case study of a marine seismic acquisition system, which was developed by a company leading in this industry sector. We only ran the best combination because executing test cases on the industrial case study is very time consuming and we could not, for technical reasons, run it on a cluster. We compared the performance of RT and this heuristic combination when used with GA and (1+1)EA on the industrial case study.

The rest of the paper is organized as follows: Section 2 provides a background of the work. Section 3 discusses related work. Section 4 provides an introduction to the earlier proposed environment modeling methodology and testing approach. Section 5 discusses the new search heuristics, whereas Section 6 discusses the empirical study carried out to evaluate the new search heuristics. Finally, Section 7 concludes the paper.

2. Background

Several software engineering problems can be reformulated as a search problem, such as test data generation [8]. An exhaustive evaluation of the entire search space (i.e., the domain of all possible combinations of problem variables) is usually not feasible. There is a need for techniques that are able to produce “good” solutions in reasonable time by evaluating only a tiny fraction of the search space. Search algorithms can be used to address this type of

problem. Several successful results by using search algorithms are reported in the literature for many types of software engineering problems [9].

To use a search algorithm, typically a fitness function needs to be defined that is used to guide the search algorithms toward fitter solutions. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. Eventually, given enough time, a search algorithm will find a satisfactory solution.

There are several types of search algorithms. Genetic Algorithms (GA) are the most well-known [9], and they are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through crossover and mutation operators. (1+1) Evolutionary Algorithm (EA) is simpler than GAs, in which only a single individual is evolved with mutation.

To cope with several problems related to combining together different heuristics/objectives with different priorities, we rather use an *order* function h . An order function takes two solutions as parameters and returns whether the first is better, equivalent, or worse than the second solution (e.g., by returning 1, 0, and -1 respectively). For a search algorithm, an order function h can always replace a fitness function f as long as the raw fitness values are not used besides comparing solutions' fitness. For example, h can be used in a GA using tournament or rank selection, but not for fitness proportional selection. For more details, examples and discussions regarding order functions for search algorithms in software testing can be found in [10].

3. Related Work

Depending on the goals, testing of RTES can be performed at different levels: model-in-the-loop, hardware-in-the-loop, processor-in-the-loop, and software-in-the-loop [11]. Our approach falls in the software-in-the-loop testing category, in which the embedded software is tested on the development platform with a simulated environment. The only variation is that, rather than simulating the hardware platform, we use an adapter that forwards the signals from the system under test (SUT) to the simulated environment. This helps focus on testing the

embedded software. This approach is especially helpful when the software is to be deployed on multiple hardware platforms or the target hardware platform is stable (such as the case with our industry partners, working in the area of marine seismic acquisition and automated bottle recycling machines).

A large body of research has been carried out for RTES testing. Most of these approaches are based on testing the violation of timing constraints [12] or checking their conformance to a specification [13]. The specification is generally a formal model of the system and this model is then used to generate the test cases. As specification of the system, a number of approaches use Timed Automata or one of its extensions (e.g., [14]). For the same purpose, UML statechart [15], Extended Finite State Machines [16] and Attributed Event Grammar [17] have also been used. There are also several works using search-based testing techniques for testing different aspects of RTES, as for example identify deadline misses [18]. Most of the work on search-based software testing has been focused on unit testing [19], and not system level testing as we do in this paper.

There are also a few works discussing RTES testing based on environment models rather than system models. Auguston *et al.* [17] discusses the development of environment behavioral models using an event grammar for testing of RTES. The behavioral models contain details about the interactions with the SUT and possible hazardous situations in the environment. Heisel *et al.* [20] propose the use of a requirement model and an environment model along with the model of the SUT for testing. Adjir *et al.* [21] discuss a technique for testing RTES based on the system model and assumptions in the environment using Labeled Prioritized Timed Petri Nets. Larsen *et al.* [22] propose an approach for online RTES testing based on time automata to model the SUT and environmental constraints. Peleska *et al.* [23] present a benchmark model for testing RTES in the automotive domain. Their testing methodology uses information from environment models and system models to obtain test cases.

The work presented here is significantly different from most the above approaches as we adopt, for practical reasons, a black-box approach to system testing that relies exclusively on modeling the RTES environment rather than its internal design properties. This is of practical importance as independent system test teams usually do not have easy access to precise design information. Most existing works do not focus on system testing, hence their emphasis is on

modeling the RTES internal behavior and structure. Another difference of practical importance, though this is not in the focus of this paper, is that we use UML and its standard extensions for modeling the environment [5].

4. Environment Modeling and Model-based Testing

This section introduces our previous work on which we build in this paper.

4.1 Environment Modeling & Simulation

For RTES system testing, as we observed among our industry partners, software engineers familiar with the application domain would typically be responsible for developing the environment models. Therefore, we selected UML and its extensions as the environment modeling language. As a standard modeling language, it is widely taught and accepted by software engineers and supported by a broad range of tools and training material, all of which being important considerations for successful industry adoption.

The environment models consist of a domain model and several behavioral models. The domain model captures the structural details of the RTES environment, such as the environment components, their relationships, and their characteristics. The behavior of the environment components is captured by state machines. These models are developed, based on our earlier proposed methodology by using UML, MARTE, and our proposed profile for environment modeling [5]. These models not only include the nominal functional behavior of the environment components (e.g., booting of a component) but also include their robustness (failure) behavior (e.g., break down of a sensor). The latter are modeled as *failure* states in the environment models. The behavioral models also capture what we call *error* states. These are the states of the environment that should never be reached if the SUT is implemented correctly (e.g., no incorrect or untimely message from the SUT to the environment components). Therefore, error states act as oracles for the test cases.

An important feature of these environment models is that they capture the non-determinism in the environment, which is a common characteristic for most RTES environments. Non-determinism may include, for example, different occurrence rates and patterns of signals, failures of components, or user commands. The environment modeling profile provides special constructs to model non-deterministic behavior of the environment. Each environment

component can have a number of non-deterministic choices whose exact values are selected at the time of testing. Java is used as an action language and OCL (Object Constraint Language) is used to specify constraints and guards. In general, for the type of system testing we do, a communication layer is needed to make the simulated environment communicate with the actual RTES (e.g., to receive stimuli and to send responses). Such a communication layer is written by the software engineer separately from the models. This allows for the simulators and models to be independent of the language in which SUT is written.

Using model to text transformations, the environment models are automatically transformed into environment simulators implemented in Java. The transformations follow specific rules that we discussed in detail in [6]. During simulation a number of instances can be created for each environment component, which interact with each other and the SUT (for example multiple instances of a sensor component). The generated simulators are linked with the test framework that provides the appropriate values for each simulation execution. For all our case studies, the generated simulators communicate with the SUT using TCP sockets. The choice of Java and TCP is based on actual requirements of one of our industrial partners, where the RTES under study involves soft real-time constraints.

Environment simulation is an important feature for the type of testing that we do. Our target systems are typically reactive systems and depending on their internal states, they may behave differently to the same environment stimuli. Therefore, in some cases, the exact response from the SUT to a particular environment event cannot be determined before execution. Environment models are developed in a way that they accept different responses of the SUT that may be triggered as a result of the environment events, including invalid responses that lead to error states. The simulation allows the environment to handle such non-determinism in the SUT, since depending on the response of SUT, the environment can simulate any of the modeled behavior.

4.2 Environment Model-Based Testing

In our context, a test case execution is akin to executing the environment simulator. The domain model represents various components in the RTES environment. As mentioned earlier, during a simulation there can be multiple instances for each of the environment components and multiple components run in parallel to form the RTES environment. During the simulation, values are required for the non-deterministic choices in the environment models. A

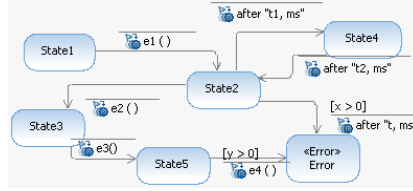


Figure 1. A dummy state machine to explain search heuristics

test case in our context provides information for both the number of instances for each component (which we refer to as the environment configuration) and the values for various non-deterministic choices (referred to as the simulation configuration). For the scope of this paper, we only consider one fixed environment configuration; therefore in the rest of the paper, a test case is alternatively used for referring to a simulation configuration.

A test case can be seen as a test data matrix, where each row provides a series of values for a non-deterministic choice of the environment component (the number of rows is equal to the number of non-deterministic choices). Each time a non-deterministic choice needs to be made, a value from the corresponding matrix row is selected. During simulation, a query for a non-deterministic choice can be made several times and the number of queries cannot be determined before simulation. To resolve this problem, each matrix row (a data vector) can be represented in two possible forms: a fixed length ring or a variable length vector. On one hand, in the fixed-length ring vector, the vector is considered as a ring and upon reaching the end/tail of the vector. Then, the values are again selected from the start/head of the vector. On the other hand, in the variable size vector, whenever the end of a vector is reached, its size is increased at run time and new values are added. In our earlier work [2], we evaluated the effect of the representations and starting lengths of the test data vectors on the fault detection effectiveness.

In our earlier work, we applied various testing strategies to generate test cases from the environment models [1]. For search-based testing, we developed a new fitness function f that can be seen as an extension of the fitness function developed for model-based testing based on system specifications [24]. The original fitness function uses the so-called “approach level” and normalized “branch distance” to evaluate the fitness of a test case. For environment model-based testing, we introduced the novel concept of normalized “time distance”. In our context, the goal is to minimize the fitness function f , which heuristically evaluates how far a

test case is from reaching an error state. If a test case with test data m is executed and an error state of the environment model is reached, then $f(m) = 0$.

The approach level (A) refers to the minimum number of transitions in the state machine that are required to reach the error state from the closest executed state. Figure 1 shows a dummy example state machine to elaborate the concept. The state named *Error* is the error state. Events $e1$, $e2$, and $e3$ are signal events, whereas events *after* “ t, s ”, *after* “ $t1, ms$ ”, and *after* “ $t2, ms$ ” are time events with t , $t1$, and $t2$ as the time values and ms and s as time units. Events $e3$ and *after* “ t, s ” are guarded by constraints using OCL. If the desired state is *Error* and the closest executed state was *State5*, then the approach level is 1.

The approach level rewards test case executions that get closer to an error state, but it does not provide any gradient (guidance) to solve the possible guards on the state transitions. The branch distance (B) is used to heuristically score the evaluation of the guards (if any) on the outgoing transitions from the closest executed state. In [25] we have defined a specific branch distance function for OCL expressions that is reused here for calculating the branch distance. In the dummy state machine in Figure 1, we need to solve the guard “ $y > 0$ ” so that whenever $e4$ is triggered, then the simulation can transition to the *Error* state. Note that branch distance is less important than approach level, since it is required only when the transition towards an error state is guarded and the approach level cannot be reduced any further. Therefore, we normalized the branch distance in the range of 0 to 1 [10].

The third important part of the fitness function is the time distance (T), which comes into play when there are timeout transitions in the environment models. For example, in Figure 1, the transition from *State2* to *Error* is a timeout transition. If a transition should be taken after z time units, but it is not, we calculate the maximum consecutive time c the component stayed in the source state of this transition (e.g., *State2* in Figure 1). To guide the search, we use the following heuristic: $T = z - c$, where $c \leq z$. Again, the importance of time distance is less than that of approach level, therefore it is normalized in the range 0 to 1. The fitness function f using these three heuristics for a test data matrix m is defined as:

$$f(m) = \min_e ((A_e(m) + \text{nor}(T_e(m)) + \text{nor}(B_e(m)))) \quad (1)$$

where for an error state e , A_e represents the approach level, T_e represents the time distance, and B_e represents the branch distance. $\text{nor}()$ is the normalizing function. For guarded time transitions, B_e was only calculated after the corresponding time event was triggered. Since,

there can be multiple error states in the environment models, the function $f(m)$ only takes the minimum value over all error states (represented by min_e in (1)).

The results when using this fitness function, as reported in [1], were disappointing. The branch distance was calculated for the guards only after an event was triggered and this worked fine for signal events. But for time events, this meant that to get the branch distance, we first needed to trigger the time event. For this we focused first on reducing the time distance and then calculated the branch distance. It turned out that this assumption of favoring reduction of time distance whenever there is a time transition was naive. In situations where the time transition had a guard, a test case with less time distance but with a greater branch distance was considered to be better than a test case with greater time distance but lower branch distance. However, there is no purpose in reducing the time distance (i.e., the error state will not be reached) if at the end the transition is not fired because the guard is false.

5. Improved Fitness Function

In this section, we present novel improvements in the fitness function f for environment model-based testing of RTES. As mentioned earlier, for problems related to combining various heuristics/objectives with different priorities, we can replace the use of a fitness function f with an order function h . For two test data matrices m_1 and m_2 , the function h will return 1, 0, or -1 if m_1 is better, equal, or worse than m_2 , respectively.

Following, based on $f(m)$ we define a basic order function h for two test data matrices (m_1 , m_2) that will be reused for definition of order functions for the three new heuristics: Time In Risky State (TIR), Risky State Count (RSC), and Coverage (COV).

$$h(m_1, m_2) = \begin{cases} 1 & \text{if } A_{\min}(m_1) < A_{\min}(m_2) \text{ or } (A_{\min}(m_1) = A_{\min}(m_2) \text{ and} \\ & B_{\min}(m_1) < B_{\min}(m_2)) \text{ or } (A_{\min}(m_1) = A_{\min}(m_2) \text{ and} \\ & B_{\min}(m_1) = B_{\min}(m_2) \text{ and } T_{\min}(m_1) < T_{\min}(m_2)) \\ 0 & \text{if } A_{\min}(m_1) = A_{\min}(m_2) \text{ and } B_{\min}(m_1) = B_{\min}(m_2) \text{ and} \\ & T_{\min}(m_1) = T_{\min}(m_2) \\ -1 & \text{otherwise} \end{cases} \quad (2)$$

where for a set of error states es , $A_{\min}(m)$ is defined as the minimum approach level for the matrix m over es , $B_{\min}(m)$ as the minimum branch distance for m over es , and $T_{\min}(m)$ as minimum time distance for m over es . A_{\min} takes precedence on B_{\min} and T_{\min} , and B_{\min} takes precedence on T_{\min} . This is simply reflecting the relative importance of these three heuristics.

5.1 Improved Time Distance (ITD)

We improved the way the basic time distance was calculated in the earlier fitness function. The motivation behind the improved time distance is that to avoid fitness plateaus, a test case with a lower branch distance for a time transition should be preferred over the one having greater branch distance, irrespective of the time distance. This is due to the fact that during environment simulation, changing the values of a test case often has a direct impact on the time distance and it should therefore be easier to reduce it than the branch distance. For example in Figure 1, the time transition *after* “ t, s ” is guarded by $[x > 0]$. A test case with a positive value greater than 0 for x will be considered better than a test case with a negative or 0 value for x , irrespective of the value of t . The value of t is considered only after the branch distance of the guard equals 0. For this, we introduced the concept of a look-ahead branch distance (LB) for time transitions, which represents the branch distance of OCL guard on a time transition when it is not fired (i.e., the timeout did not occur). Because OCL evaluations are free from side-effects [25], this does not lead to any particular problem. The order function for two test data matrices m_1 and m_2 using this heuristic is:

$$\begin{aligned}
 h(m_1, m_2) &= \begin{cases} \mathbf{1} & \text{if } A_{\min}(m_1) < A_{\min}(m_2) \text{ or } (A_{\min}(m_1) = A_{\min}(m_2) \text{ and } B_{\min}(m_1) < B_{\min}(m_2)) \text{ or } (A_{\min}(m_1) = A_{\min}(m_2) \text{ and } B_{\min}(m_1) = B_{\min}(m_2) \text{ and } ITD_{\min}(m_1, m_2) = 1) \\ \mathbf{0} & \text{if } A_{\min}(m_1) = A_{\min}(m_2) \text{ and } B_{\min}(m_1) = B_{\min}(m_2) \text{ and } ITD_{\min}(m_1, m_2) = 0 \\ \mathbf{-1} & \text{otherwise} \end{cases} \quad (3) \\
 ITD_e(m_1, m_2) &= \begin{cases} \mathbf{1} & \text{if } LB_e(m_1) < LB_e(m_2) \text{ or } (LB_e(m_1) = LB_e(m_2) \text{ and } T_e(m_1) < T_e(m_2)) \\ \mathbf{0} & \text{if } (LB_e(m_1) = LB_e(m_2) \text{ and } T_e(m_1) = T_e(m_2)) \\ \mathbf{-1} & \text{otherwise} \end{cases}
 \end{aligned}$$

where for the set of error states es and a given error state $e \in es$, $A_{\min}(m)$ represents the minimum approach level for matrix m over es , $B_{\min}(m)$ is the minimum branch distance for m over es , $LB_e(m)$ represents the look-ahead branch distance for m for the error state e , and $T_e(m)$ represents the time distance for m over e .

5.2 Time in Risky State (TIR)

A “risky state” is defined as a state adjacent to the error state (i.e., approach level = 1). For the order function, when two test cases have the same A_{min} , B_{min} , and T_{min} , then a test case that spends more time in risky states should have higher fitness. The motivation behind this heuristic is that, the more time spent in a risky state, the higher the chances of events happening in the environment or SUT leading to the error state (e.g., receive a signal from the SUT). For example, for the state machine shown in Figure 1, this heuristic will favor the test cases that spend more time in the risky states *State2* or *State5*. For instance in *State2*, it is possible to increase the value of $t1$ in the time event *after* “ $t1, ms$ ”, which will increase the time spent in this state. The overall order function based on h defined in (2), is given as:

$$h'(m_1, m_2) = \begin{cases} h(m_1, m_2) & \text{if } h(m_1, m_2) \neq 0 \\ \mathbf{1} & \text{if } h(m_1, m_2) = 0 \text{ and } TIR_{sum}(m_1) > TIR_{sum}(m_2) \\ \mathbf{0} & \text{if } h(m_1, m_2) = 0 \text{ and } TIR_{sum}(m_1) = TIR_{sum}(m_2) \\ \mathbf{-1} & \text{otherwise} \end{cases}$$

where $TIR_{sum}(m)$ is the sum of time spent in risky states for all error states and the test data matrix m .

5.3 Risky State Count (RSC)

This heuristic is also based on utilizing the concept of risky states: When two test cases have the same A_{min} , B_{min} , and T_{min} , then a test case that enters a risky state more often should be preferred over a test case that does so less often. For example, for the state machine shown in Figure 1, this heuristic will assign higher fitness to the test cases that make the component enter *State2* more often, i.e., transitions to *State4* and come back. This would for instance result in minimizing the values of $t1$ and $t2$ for the timeout transitions *after* “ $t1,s$ ” and *after* “ $t2,s$ ” to increase the risky state count. Note that the heuristic will only be useful for the cases that allow a loop back to a risky state. The overall order function based on the basic order function h defined in (2) is:

$$h'(m_1, m_2) = \begin{cases} h(m_1, m_2) & \text{if } h(m_1, m_2) \neq 0 \\ \mathbf{1} & \text{if } h(m_1, m_2) = 0 \text{ and } RSC_{sum}(m_1) > RSC_{sum}(m_2) \\ \mathbf{0} & \text{if } h(m_1, m_2) = 0 \text{ and } RSC_{sum}(m_1) = RSC_{sum}(m_2) \\ \mathbf{-1} & \text{otherwise} \end{cases}$$

where $RSC_{sum}(m)$ is total count of transitions made to all risky states for the test data matrix m .

5.4 Increase in Coverage (COV)

This heuristic is based on the concept of coverage of environment models. This heuristic, when two test cases have the same A_{min} , B_{min} , and T_{min} , calculates the environment coverage and assign higher fitness to the test cases that cover more environment states.

The idea behind this heuristic is to increase the coverage of the environment models when the approach level, branch distance and time distance can no longer be improved. The assumption is that having higher environment coverage will result in more diversity in the test cases, which might lead to situations that help reach the error state. For example in Figure 1, this heuristic will favor a test case that visited *State4* over a test case that did not. The idea is to explore more states and transitions in the environment models. The overall order function for *COV* based on h (2) is:

$$h'(m_1, m_2) = \begin{cases} h(m_1, m_2) & \text{if } h(m_1, m_2) \neq 0 \\ 1 & \text{if } h(m_1, m_2) = 0 \text{ and } COV_{min}(m_1) > COV_{min}(m_2) \\ 0 & \text{if } h(m_1, m_2) = 0 \text{ and } COV_{min}(m_1) = COV_{min}(m_2) \\ -1 & \text{otherwise} \end{cases}$$

where $COV_{sum}(m)$ is the total coverage for all error states.

5.5 Combination of heuristics

Apart from the individual heuristics, we also investigate their combinations. In total, for the latter three heuristics (*TIR*, *RSC*, and *COV*) there are eight possible combinations. They can be combined with the basic order function h and an order function containing the improved time distance ITD instead of T in h , which results in a total of 16 possible combinations

$$h'(m_1, m_2) = \begin{cases} h(m_1, m_2) & \text{if } h(m_1, m_2) \neq 0 \\ 1 & \text{if } h(m_1, m_2) = 0 \text{ and } comb(m_1) > comb(m_2) \\ 0 & \text{if } h(m_1, m_2) = 0 \text{ and } comb(m_1) = comb(m_2) \\ -1 & \text{otherwise} \end{cases}$$

where $comb(m)$ is a given combination of the heuristics.

When combining these heuristics, we follow the Pareto dominance principle - a key concept for multi-objective optimization in evolutionary algorithms [26]. In our context this means that, given a combination of heuristics, a test data matrix m_1 will dominate another

matrix m_2 , if it is better than m_2 for at least one heuristic and is not worse than m_2 in any of the other heuristics. The reasons for using a Pareto dominance is that, in contrast to approach level and branch distance, we do not know which is the most important heuristic among the three that were proposed: this is a research question that we address in this paper.

6. Empirical Study

The objective of this empirical study is to evaluate the effectiveness, in terms of fault detection, of the proposed heuristics and their combinations. We selected two search algorithms for this empirical study: Genetic Algorithms (GA) and (1+1)Evolutionary Algorithm (EA). Though (1+1) EA is simpler than GA, it has shown better results in our previous testing works (e.g., [25]). We use the convention *Algorithm-Heuristic* to denote an algorithm using a heuristic or its combination. For example, to denote that GA is used with the basic fitness function defined in (1), we use the terms GA-Basic.

6.1 Case Study

For the sake of experimenting with diverse environment models and RTES, we developed 13 different artificial RTES that were inspired by two industrial cases we have been involved with [5] and one case study discussed in the literature [16]. Since, there are no benchmark RTES available to researchers, we specifically designed these artificial problems to conduct our experiments (called AP1 – AP13). The goal while developing the models of these RTES was to vary various characteristics of the environment models (e.g., guarded time transitions, loops) in order to evaluate the impact of these characteristics on the test heuristics. We could not have covered such variations in environment models with one or even a few industrial case studies, hence the motivation to develop artificial cases. Nine of these artificial problems were inspired by a marine seismic acquisition system developed by one of our industrial partners. These problems covered various subsets of the environment of the industrial RTES. Three of the 13 problems were inspired by the behavior of another industrial RTES (part of an automated recycling machine) developed by another industrial partner. The thirteenth artificial problem was inspired by the train control gate system described in [16].

The industrial case study we also report on is a very large and complex seismic acquisition system that interacts with several sensors and actuators. The timing deadlines on the

environment are in the order of hundreds of milliseconds. The company that provided the system is a market leader in its field. For confidentiality reasons we cannot provide full details of the system. The SUT consists of two processes running in parallel, requiring a high performance, dedicated machine to run.

Table 1. Summary of environment models*

Problem	Guard on Path	Time transition on Path	Loop to Risky State	Guard on Error Transition	Tine Transition to Error State	Approach to Risky State
AP1	Yes	Yes	No	Yes	Yes	Non-trivial
AP2	Yes	Yes	No	Yes	Yes	Non-trivial
AP3	No	Yes	No	No	Yes	Non-trivial
AP4	No	Yes	No	No	Yes	Non-trivial
AP5	No	Yes	No	No	Yes	Non-trivial
AP6	Yes	Yes	Yes	Yes	Yes	Non-trivial
AP7	Yes	Yes	Yes	Yes	Yes	Non-trivial
AP8	Yes	Yes	Yes	Yes	Yes	Non-trivial
AP9	No	No	Yes	No	No	Trivial
AP10	Yes	Yes	Yes	Yes	Yes	Trivial
AP11	Yes	Yes	No	Yes	Yes	Trivial
AP12	Yes	Yes	No	Yes	Yes	Trivial
AP13	Yes	Yes	No	Yes	Yes	Trivial
IC	Yes	Yes	Yes	Yes	Yes	Trivial

To facilitate the discussion of our results, a summary of relevant characteristics for the environment models of the RTES under study is provided in Table 1. The columns ‘Guard on Path’ and ‘Time transition on Path’ represent whether these features were present on a path to the error state. The column ‘Loop to Risky state’ reports whether there was a loop back to a risky state (i.e., an outgoing transition to a state and then returning back to the risky state). The columns ‘Guard to Error Transition’ and ‘Time transition to Error’ show whether these features were present on the transition from the risky state to the error state. The column ‘Approach’ shows if the approach to the risky state (i.e., obtaining a test case in which the closest executed state is the risky state) is trivial or not. It is considered to be trivial if a risky state is reached on average by the first ten randomly executed test cases. The row in Table 1 with problem IC summarizes the characteristics of the environment models for the industrial case study.

These RTES are written in Java to facilitate their use on different machines and operating systems. The communication between the RTES and their environments is carried out through

TCP. All these RTES are multithreaded. Each of the artificial problems had one error state in their environment models and non-trivial faults were introduced by hand in each of them. We could have rather seeded those faults in a systematic way, as for example by using a mutation testing tool [27]. We did not follow such procedure because the SUTs are highly multi-threaded and use a high number of network features (e.g., opening and reading/writing from TCP sockets), which could be a problem for current mutation testing tools. Furthermore, our testing is taking place at the system level, and though small modifications made by a mutation testing tool might be representative of faults at the unit level, it is unlikely to be the case at the system level for RTES. On the other hand, the faults that we manually seeded came from our experience with the industrial RTES and from the feedback of our industry partners. For the industrial case study, we did not seed any fault and the goal was to find the real fault that we initially uncovered in [1].

Table 2. Success rates of various heuristic for GA & EA

Problem	Basic		ITD		TIR		RSC		COV	
	GA	EA	GA	EA	GA	EA	GA	EA	GA	EA
AP1	0.3	0	0.05	0.15	0.9	1	0.2	0.05	0.4	0
AP2	0.65	0.3	0.5	0.5	11	1	0.65	0.55	0.6	0.25
AP3	0.4	1	0.5	0.9	0.5	0.9	0.45	0.9	0.5	1
AP4	0.9	1	0.95	1	1	1	0.95	1	0.95	0.95
AP5	0	0.55	0.05	0.6	0.05	0.95	0.05	0.5	0.05	0.6
AP6	0.65	0.5	0.85	0.9	0.65	0.75	0.4	0.35	0.5	0.15
AP7	1	0.9	1	0.9	1	1	0.95	0.9	0.95	0.3
AP8	0.15	0.1	0.15	0.55	0	0.3	0	0.05	0	0.05
AP9	0.75	0.65	0.8	0.45	0.6	0.4	0.9	1	0.45	0.45
AP10	1	0.9	1	0.85	1	0.9	1	0.95	0.85	0.15
AP11	0.55	0.75	0.75	0.8	0.6	0.7	0.65	0.45	0.65	0.45
AP12	0.25	0.25	0.3	0.1	0.25	0.05	0.25	0	0.15	0.1
AP13	0.95	1	1	1	0.85	0.9	0.95	0.85	1	0.75
Average	0.58	0.61	0.61	0.67	0.65	0.76	0.57	0.58	0.54	0.4

6.2 Experiments

In this paper, we want to answer the following research questions: **RQ1:** What is the effect on fault detection of new order functions having each one of the proposed heuristics: Improved Time Distance (ITD), Time In Risky State (TIR), Risky State Count (RSC), and Coverage (COV) compared to the previously defined basic fitness function for GA and (1+1) EA? **RQ2:** Which combinations of the proposed heuristics are best in terms of fault detection? **RQ3:**

Between the two search-based algorithms, GA and (1+1) EA, which one works better in terms of fault detection with the new heuristics? **RQ4:** How do the search-based algorithms compare to random testing (RT)? **RQ5:** How does the best combination of the proposed heuristics compare to RT, GA-Basic, and (1+1) EA-Basic on the industrial case study?

To answer the research questions RQ1 – RQ4, we carried out a series of experiments on the above-mentioned thirteen artificial problems. For RQ5, we conducted the experiments on the industrial case study. We ran two search algorithms, (1+1) EA and GA, to answer these research questions. We also used RT as a comparison baseline for RQ2, RQ4, and RQ5, as it is the simplest solution to implement. For GA, we employ rank selection with bias 1.5 to choose the parents, the initial population size is 10 and a single point crossover is used with probability $P_{\text{crossover}} = 0.75$. Different settings of these parameters could lead to different performance, but we selected reasonable parameter values following recommendations in the GA literature [28].

For the experiments, we ran RT, GA, (1+1) EA on each of the 13 problems. We have three order functions for the individual heuristics and can combine them in 12 different ways (as described in Section 5.5). We ran these combinations with both the basic order function (defined in (2)) and the order function using ITD (defined in (3)). In total we therefore executed $2 * (8 * 2) * 13 + 13 = 429$ experiment configurations (two search algorithms, 16 order functions, 13 artificial problems, on which RT is also run). The execution time of each test case was fixed to 10 seconds and we stopped each algorithm after 1000 sampled test cases or as soon as we reached any of the error states. The choice of running each test case for 10 seconds was based on the properties of the RTES and the environment models. The objective was to allow enough time for the test cases to reach an error state. For each of these 429 experiment configurations, we ran each algorithm 20 times with different random seeds. The total number of sampled test cases was 7,676,635, which required around 888 days of CPU resources. Therefore, we performed the experiments on a cluster of computers.

To answer the research question RQ5, we carried out experiments on the industrial case study. We run each test case for 60 seconds, where 1000 test case executions (fitness evaluations) can take more than 16 hours. This choice has been made based on the properties of the RTES and discussions with the actual testers. Due to the large amount of resources required, we only ran the combination of heuristics that on average gave best results for the

thirteen artificial problems. We compared its fault detection rate with that of GA-Basic, (1+1) EA-Basic, and RT. We carried out 20 runs for each of these four experiment configurations. The total number of sampled test cases was 42,073, which required over 29 days of computation on a single, high-performance, dedicated machine.

To analyze the results, we used the guidelines described in [29] which recommends a number of statistical procedures to assess randomized test strategies. First we calculated the success rates of each algorithm: the number of times it was successful in reaching the error state out of the total number of runs. These success rates are then compared using the Fisher Exact test, quantifying the effect size using an odds ratio (ψ) with a 0.5 correction (p-values of this test are denoted as p in the tables showing the results). When the differences between the success rates of two algorithms were not significant, we then looked at the average number of test cases that each of the algorithms executed to reach the error state. We used the Mann-Whitney U-test and quantified the effect size with the Vargha-Delaney A_{12} statistics (p-values of this test are denoted as $it-p$ in the tables showing the results). The significance level for these statistical tests was set to 0.05. In all the tables showing the odds ratio and A_{12} statistics, when comparing two algorithms, say q and r , a bold-faced font shows that q is significantly better than r and an italicized font shows that q is significantly worse than r . Table cells with a ‘-’ denote no significant results for the comparison.

Table 3. Results of ITD compared with basic fitness function

Problem	GA-ITD vs. GA	EA-ITD vs. EA
AP6	-	p=0.0138, ψ =7.4
AP8	-	p=0.0057, ψ =8.96

Table 4. Results of TIR compared with basic fitness function

Problem	GA-TIR vs. GA	EA-TIR vs. EA
AP1	p= 0.00024, ψ = 16.51	p=1.45e-11, ψ =1681
AP2	p= 0.00832, ψ = 22.78	p=3.34e-06, ψ = 91.46
AP3	-	it-p = 0.00167, A_{12} = 0.8
AP5	-	p=0.00836, ψ = 10.74
AP6	<i>it-p= 0.03125, A_{12} = 0.24</i>	-
AP10	-	it-p= 0.02677, A_{12} = 0.7

6.3 Results and Discussion

We decompose RQ1 into four sub questions (RQ1a - RQ1d), one for each heuristic. Table 2 shows the success rates for the 13 artificial problems and the four heuristics with GA and (1+1) EA.

Results when applying ITD (RQ1a) to the artificial problems with GA and EA are shown in Table 3 and are compared with results obtained when using the basic fitness function. The table shows the p-values and odds ratio when success rates were significantly different and otherwise, the p-value and the A_{12} statistics on the difference in the number of test case executions to reach the error state. Using ITD with (1+1) EA yields significantly better results for two of the artificial problems. In other cases the performance of the algorithm with this order function was the same as that for the basic algorithm. ITD relies on information regarding guarded time transitions in the models. Among the thirteen artificial problems, AP3 – AP5 and AP9 did not have any guard or time transition leading to the error state. Even in these cases, ITD shows similar performance to basic fitness with no significant drawbacks. To answer RQ1a, using the fitness function with ITD can bring improvements in fault detection effectiveness for (1+1) EA and has no significant difference when used with GA.

Turning now to Table 4, when TIR was used with GA (RQ1b), it gave significantly better results in two of the artificial problems and was worse in one problem (AP6). For other artificial problems, the results of the two algorithms were comparable. When TIR was used with (1+1) EA, it gave significantly better results for five of the 13 artificial problems. In other cases there were no significant differences. To answer RQ1b, TIR performs better or similar to the basic fitness for all but one of the artificial problems, whereas the performance of TIR with EA is better or equal to the (1+1) EA-Basic in all the cases. Hence the use of TIR in the order function seems to be an effective option in most cases.

Table 5 addresses RQ1c and evaluates the RSC heuristic. When RSC was used with GA, it gave significantly better results in one of the artificial problems (AP10) and showed no significant difference for the other artificial problems. When RSC was used with (1+1) EA, it gave significantly better results for one artificial problem (AP9), worse results for another one (AP12), and no statistical differences otherwise. RSC depends on the presence of a loop back to a risky state. According to the information in Table 1, AP6 – AP10 had a loop back to the risky state. Hence, we can answer RQ1c by stating that for all the problems that have a loop to risky states, an order function using the RSC heuristic performs significantly better or similar to the basic fitness function. But for the problems without such a loop, it can negatively affect performance. Table 6 addresses RQ1d and evaluates the Coverage (COV) heuristic. When COV was used with GA, there were no statistical differences between the results. When it was

used with (1+1) EA, it gave significantly worse results for four of the artificial problems and yielded no significant differences in other cases. To answer RQ1d, using the order function with coverage only can result in significant deterioration in the performance of (1+1) EA.

To summarize the comparison of proposed heuristics with basic fitness (RQ1), we can state that ITD and TIR heuristics shows significant improvements for (1+1) EA and in most cases for GA. RSC shows improvements in cases where there is a loop to risky states, otherwise it can negatively affect the performance. Finally the COV heuristic shows worse performance for (1+1) EA and no difference for GA.

Next, we answer RQ2, for which we evaluate the various combinations of the four proposed heuristics. As discussed we had a total of 16 possible order functions for each search algorithm. Table 7 provides the relative ranking based on the statistical difference of the compared configurations. Configurations which are statistically equivalent (i.e., p-values above 0.05) are expected to show a similar ranking. This is done by assigning scores based on pairwise comparisons of configurations. Whenever a configuration is better than the other and the difference is statistically significant, its score is increased. Then, based on the final scores, each configuration is assigned ranks ranging from 1 (best configuration) to 33 (worst configuration). In case of ties, ranks are averaged. The configurations in the table are sorted by their average ranking (last column) in an ascending order.

Overall, based on the average ranks for the 13 artificial problems, (1+1) EA with TIR proved to be the best algorithm for both Basic and ITD versions of the heuristic. Analyzing the results of Table 7 according to the characteristics of artificial problem, we can conclude that in general search-based algorithms perform significantly worse than RT for the artificial problems where the approach to risky states is trivial (see discussion for RQ4 and a plausible detailed explanation at the end of this section). If we exclude the results of such artificial problems (i.e., AP9 – AP13), then in all the other problems, (1+1) EA with ITD and TIR performed significantly better than other combinations. According to the ranks shown, the only exception seems to be AP7, but even in that case, though the number of test case executions is significantly less for other order functions, the success rate of (1+1) EA with both the order functions (Basic-TIR and ITD-TIR) was 100%. If we only consider GA, then the best two algorithms were GA-ITD-TIR and GA-ITD-TIR-RSC. The good overall performance of TIR is likely to be due to the fact that it focuses on making the environment

spend more time in the risky states, thus increasing the occurrence of situations that lead to the error state. When we compared the performance of (1+1) EA-Basic-TIR with (1+1) EA-ITD-TIR, there were no significant differences in the results. But looking at the results in Table 7, where for various combinations used with (1+1) EA-ITD and (1+1) EA-Basic, the combinations used with (1+1) EA-ITD showed better or statistically equal results. This further confirms the findings of RQ1a, which suggested to use (1+1) EA-ITD over (1+1) EA-Basic.

Table 5. Results of RSC compared with basic fitness function

Problem	GA-RSC vs. GA	EA-RSC vs. EA
AP9	-	p=0.00831, ψ =22.78
AP10	it-p= 0.0073, A_{12} = 0.74	-
AP12	-	<i>p=0.047, ψ =22.78</i>

Table 6. Results of COV compared with Basic fitness function

Problem	GA-COV vs. GA	EA-COV vs. EA
AP6	-	<i>p = 0.0407, ψ = 5</i>
AP7	-	<i>p = 0.0002, ψ =16.5</i>
AP10	-	<i>p = 3.36e-06, ψ =37</i>
AP13	-	<i>p = 0.0471, ψ =14.5</i>

Regarding RQ3 (about the comparison of GA and (1+1) EA), based on Table 7, (1+1) EA seems overall to provide significantly better results with various combinations when compared to GA using the same combinations of heuristics. An exception to this is when EA is used with the coverage heuristic, in which case it performs significantly worse than GA. Even for the problems with non-trivial approach level, the performance of most of the heuristic combinations for EA is significantly better than their performance with GA. Hence, we can conclude that the fault detection effectiveness of (1+1) EA is higher than that of GA for the kind RTES system testing we focus on.

To answer RQ4 (comparison of RT with EA and GA), we compare RT with the heuristic combinations giving the best results for GA and EA. According to RQ3, for (1+1) EA, EA-ITD-TIR and EA-Basic-TIR and for GA, GA-ITD-TIR and GA-ITD-TIR-RSC were the best combinations. Table 8 shows a comparison of RT with these four algorithms. The statistics for the situations where RT is significantly worse than these algorithms are bold faced and the situations where it is significantly better are italicized. It can be observed that for all the artificial problems that have a trivial approach level (Table 1: AP9–AP13), RT performs significantly better than both search algorithms. But in other cases, where the approach level is

hard, EA and GA perform significantly better. This is especially true for EA who performs better in all the other problems, except AP7. For AP7, over 90% of the heuristics combinations had a 100% success rate and the remaining had a success rate of over 85%. Therefore, AP7 can also be considered to be a simple problem. Hence, we can answer RQ4 by stating that for simple problems (i.e., where the average success rate of all the algorithms is high or the approach level is trivial) RT performs significantly better than both search-algorithms, but for more difficult problems (i.e., lower success rates or non-trivial approach level), search algorithms perform significantly better. The best technique (1+1) EA-ITD-TIR has an average success rate of 73% for the 13 problems with an average number of 222 test case executions to find a fault. If we only consider the problems where approach level was non-trivial (i.e., excluding AP9 – AP13), then the average success rate is 84%. The worst success rate is 35% (AP8), which suggests that with r runs of the technique, we would achieve a success rate of $1 - (1-0.35)^r$. For example with only five runs ($r = 5$), we would obtain a success rate above 99%.

RQ5 is about comparing the best combination of heuristics with GA-basic, (1+1) EA-Basic and RT on the industrial case study. According to RQ2, the combination showing on average the best results for artificial problems was (1+1) EA-ITD-TIR. Table 9 shows the comparative results of running (1+1) EA-ITD-TIR, (1+1) EA-Basic, GA-Basic, and RT on the industrial case study. Table 10 shows the details of the results of this experiment including the average success rate (SR) and the average number of test case executions to find a fault (ATE). We can see that (1+1) EA-ITD-TIR shows significantly better performance over both GA-Basic and (1+1) EA-Basic. When compared to RT, there is no significant statistical difference. The best combination has relatively lower success rate (0.8 compared to 1 for RT), but it finds the fault with a lower, average number of test case executions (250 compared to 295 for RT). The better performance of RT can be explained by the fact that in the industrial case study, the approach level to risky state was again trivial as shown in Table 1 (i.e., on average it could be reached in less than 10 random test cases).

Following, we provide a plausible explanation as to why RT shows better performance when the approach level to risky state is trivial. The transition from a risky state to the error state represents the erroneous behavior of the SUT and will only be triggered if the interaction of the SUT with the environment was at some point incorrect. Therefore, triggering this transition is dependent on the behavior of the SUT. Once the environment reaches a risky state

and is not able to proceed to the error state, a possible option is to try to maximize the diversity in the environment behavior (e.g., by using entirely different values for the test data matrix, irrespective of their effect on the fitness). Maximizing diversity could result in execution of a behavior of the environment that causes the SUT to interact in an erroneous way which will in turn result in the transition to the error state. When the approach to risky state is trivial then we can simply use RT (or a similar technique) to try to maximize diversity, instead of using a technique like (1+1) EA that generates similar individuals (which makes it hard for search algorithms to be successful in such cases).

Table 7. Rank of each heuristic combination on 13 artificial problems (sorted by average rank)

Algorithm	AP1	AP2	AP3	AP4	AP5	AP6	AP7	AP8	AP9	AP10	AP11	AP12	AP13	Avg.
EA-Basic-TIR	2.5	2.5	4	4	1	3	26.5	8.5	31	12.5	17.5	24.5	23	12.35
EA-ITD-TIR	2.5	2.5	2	3	2.5	2	22	5.5	23.5	18.5	23.5	30	27	12.65
GA-ITD-TIR	7	5	24.5	21	26	7.5	1	25	25	2.5	10.5	3.5	12	13.12
EA-ITD-TIR-RSC	2.5	2.5	1	1	9.5	7.5	19	3.5	26	20.5	22	30	31	13.54
GA-ITD-TIR-RSC	10	9	24.5	21	26	7.5	8	16	3	4	26	17.5	4	13.58
GA-ITD-RSC-COV	17	18.5	24.5	12.5	26	13	4	21	4	9	17.5	2	14.5	14.12
RT	14.5	9	33	33	26	32	8	31	1	2.5	1	1	1	14.85
EA-ITD-TIR-RSC-COV	5	9	9	2	9.5	7.5	4	1.5	22	30	32	30	32.5	14.92
EA-ITD-RSC-COV	28	29	14	14.5	9.5	4	20	3.5	21	22	5	8.5	18.5	15.19
GA-ITD-RSC	19.5	22	24.5	17	26	22.5	12	16	19.5	9	2	5	3	15.23
GA-ITD-TIR-COV	23	13.5	24.5	26	26	10	12	16	16.5	9	7.5	8.5	6	15.27
EA-ITD	23	27	15.5	6	14.5	1	21	1.5	32	20.5	6	17.5	20.5	15.85
GA-Basic-TIR-RSC	11	9	24.5	21	26	17	8	25	8	12.5	20	17.5	10.5	16.15
EA-ITD-RSC	19.5	18.5	12	6	14.5	11	24	5.5	12	18.5	20	24.5	24.5	16.19
GA-Basic-RSC-COV	26.5	24.5	24.5	28.5	17.5	22.5	2	16	7	9	14.5	3.5	14.5	16.19
EA-Basic-TIR-RSC	2.5	2.5	9	9	5	20.5	24	10.5	29.5	16	23.5	30	29	16.23
GA-Basic-RSC	23	22	24.5	21	26	26	18	31	6	1	3.5	8.5	5	16.58
EA-ITD-TIR-COV	7	13.5	9	10.5	5	5	24	10.5	23.5	29	33	30	22	17.08
GA-Basic	18	22	24.5	24	26	14.5	8	16	14.5	16	14.5	8.5	17	17.19
GA-ITD-COV	26.5	18.5	24.5	27	17.5	14.5	15.5	16	10	14	14.5	17.5	8	17.23
GA-Basic-TIR-COV	13	18.5	24.5	30	26	17	14	25	5	5	14.5	17.5	14.5	17.27
GA-Basic-TIR	7	9	24.5	31	26	28.5	17	31	16.5	9	10.5	8.5	8	17.42
GA-ITD	30	27	24.5	28.5	26	12	8	16	12	16	10.5	8.5	10.5	17.65
GA-ITD-TIR-RSC-COV	23	15.5	24.5	25	26	20.5	12	16	14.5	24	7.5	17.5	8	18.00
GA-Basic-TIR-RSC-COV	14.5	15.5	24.5	32	26	24.5	15.5	25	12	6	10.5	17.5	14.5	18.31
EA-Basic-TIR-COV	12	9	4	17	5	28.5	29.5	25	9	31	26	24.5	27	19.04
GA-Basic-COV	16	24.5	24.5	21	26	27	4	31	28	23	3.5	17.5	2	19.08
EA-Basic-RSC-COV	23	30	13	14.5	16	17	31.5	8.5	18	26	26	17.5	20.5	20.12
EA-Basic-TIR-RSC-COV	9	9	6	12.5	2.5	30.5	31.5	31	19.5	28	31	30	32.5	21.00
EA-Basic-RSC	30	27	9	10.5	9.5	30.5	26.5	25	2	25	29	30	24.5	21.42
EA-Basic	32.5	31	15.5	8	9.5	24.5	29.5	16	27	27	30	12	18.5	21.62
EA-ITD-COV	30	32.5	4	17	9.5	19	28	7	33	32.5	20	24.5	27	21.85
EA-Basic-COV	32.5	32.5	9	6	13	33	33	25	29.5	32.5	28	17.5	30	24.73

If this is not the case and approach to risky state is not trivial, then a likely reason for not reaching the risky state is a guard on the transition and/or a time transition. The heuristics for search-based algorithms that we discussed in this paper are specifically designed to deal with these cases and are more suitable for such cases than RT. Our previous results on solving constraints written in OCL, lead us to the conclusion that search-based algorithms are an order of magnitude better than randomized algorithms for this purpose [25]. Hence, if the guard on the transition can be solved by directly changing the values of attributes of the environment components or the transition is a time transition, then our best chance is to use the search algorithms (and more specifically in our context, (1+1) EA-ITD-TIR).

From a practical standpoint, a possible solution to deal with the above mentioned situations that arise due to the nature of environment models is to apply RT at the start of testing and evaluate whether risky states are easy to reach. If this is the case, and if the OCL guard on the transition does not provide gradient (i.e., the so called *flag* problem [30]), then RT is most likely to trigger the transition to the error states compared to search algorithms (because of the reasons discussed above). In case the approach is not trivial, then one should use (1+1) EA-ITD-TIR, which is the best combination to use in the cases when there are guards on time transitions located on the path to the error state and is at the same time no worse than its corresponding Basic version (i.e., (1+1) EA-Basic-TIR). One limitation to this can be situations in which the approach level is not trivial and at the same time the transition leading to the risky state is only triggered in response to a particular SUT behavior (e.g., a guard that is set based on interactions with the SUT). This case will be similar to scenarios with a trivial approach to risky state in a way that the best chances of getting the SUT to behave in the required way are by invoking diverse environment behaviors. This, as we discussed earlier, is better done by RT than by the search algorithms with the proposed order functions. A possible solution to situations like these is to combine random testing with search-based algorithms and apply adaptive mechanisms based on the feedback from the executed test cases, which we will address in our future work.

In light of all the results and discussions, we can conclude that when applying our environment model-based testing approach in practice, one can achieve good results by combining RT and (1+1) EA-ITD-TIR. This can be done by running RT first and then, if no error state is reached within a short time, by running (1+1) EA-ITD-TIR for a few runs. Based

on the results reported in this paper, this strategy would be expected to achieve a success rate close to 100%.

6.4 Threats to validity

Although the artificial problems that we developed were based on industrial RTES and are not trivial (they are multithreaded and hundreds of lines long), these artificial problems are not necessarily representative of complex RTES. To reduce this threat, we used artificial problems inspired by three actual RTES and intentionally varied the properties of their environments in ways which could affect the search algorithms.

A typical problem when testing RTES is accurate simulation of time. Our approach focuses on RTES with soft time deadlines in the order of hundreds of milliseconds with an acceptable jitter of a few milliseconds. Therefore, we used the CPU clock to represent time. This might be unreliable if time constraints in the RTES were very tight (e.g., nanoseconds) since they could be violated because of unpredictable changes of load balance in the CPU in the presence of unrelated process executions. To be on the safe side, to evaluate whether our results are reliable, we selected a set of experiments and ran them again with exactly the same random seeds. We obtained equivalent results with a small variance of a few milliseconds, which in our context did not affect the testing results.

Table 8. Comparison of RT with best combinations of GA and (1+1)EA on artificial problems*

Problem	RT vs. GA1	RT vs. GA2	RT vs. EA1	RT vs. EA2
AP1	$p=0.0012, \psi=15.74$	-	$p = 0.0001, \psi = 49.63$	$p = 0.0001, \psi = 49.63$
AP2	-	-	$it-p = 0.002,$ $A_{12} = 0.2137$	$it-p = 0.0038,$ $A_{12} = 0.2312$
AP3	$0.0202, \psi = 18.38$	$p = 0.0005, \psi = 41$	$p = 3.3e-09, \psi = 303.40$	$p = 1.5e-11,$ $\psi = 1681.00$
AP4	-	-	$p = 0.0083, \psi = 22.78$	$p = 0.0083, \psi = 22.78$
AP5	-	-	$p = 3.0e-10, \psi = 533.00$	$p = 3.3e-06, \psi = 91.46$
AP6	$p = 1.7e-05, \psi = 27.13$	$p = 8.7e-05, \psi = 18.33$	$p = 0.0012, \psi = 10.33$	$p = 8.7e-05, \psi = 18.33$
AP7	-	-	$it-p = 0.0053,$ $A_{12} = 0.759$	$it-p = 0.0425,$ $A_{12} = 0.689$
AP8	-	-	$p = 0.0201, \psi = 18.38$	$p = 0.0083, \psi = 22.78$
AP9	$p=0.0004, \psi = 41$	$p=0.0202, \psi = 18.38$	$p = 4.5e-05, \psi = 60.29$	$p = 0.0004, \psi = 41.00$
AP10	-	-	-	$it-p = 0.0114,$ $A_{12} = 0.738$
AP11	$p = 0.0471, \psi = 14.55$	$p = 4.5e-05, \psi = 60.29$	$p = 0.0201, \psi = 18.38$	$p = 0.0001, \psi = 49.63$
AP12	$p = 1.3e-05, \psi = 73.80$	$p = 2.6e-08, \psi = 205.00$	$p = 3.0e-10, \psi = 533.00$	$p = 1.4e-11,$ $\psi = 1681.00$
AP13	-	-	$it-p = 0.0081,$ $A_{12} = 0.7528$	$p = 0.0202, \psi = 18.38$

* GA1 = GA-ITD-TIR, GA2 = GA-ITD-TIR-RSC, EA1 = EA-Basic-TIR, EA2 = EA-ITD-TIR

Table 9. Comparison of four algorithms on industrial case

Algorithm	(1+1)EA-Basic	(1+1)EA-ITD-TIR	RT	GA-Basic
(1+1)EA-Basic	×	$\psi=0.40, A_{12}=0.74$	$\psi=0.036, A_{12}=0.75$	$\psi=1.78, A_{12}=0.82$
(1+1)EA-ITD-TIR	$\psi=3.40, A_{12}=0.26$	×	$\psi=0.089, A_{12}=0.44$	$\psi=4.44, A_{12}=0.42$
RT	$\psi=27.88, A_{12}=0.25$	$\psi=11.18, A_{12}=0.56$	×	$\psi=49.63, A_{12}=0.47$
GA-Basic	$\psi=0.56, A_{12}=0.18$	$\psi=0.23, A_{12}=0.71$	$\psi=0.02, A_{12}=0.53$	×

Table 10. Details of each algorithm on the industrial case*

Algorithm	Success Rate	Avg. Fitness Evaluations	Standard Deviation	Median	Skewness	Kurtosis
(1+1)EA-Basic	0.6	559	270.18	615.5	-0.8	3.03
(1+1)EA-ITD-TIR	0.8	250.12	235.44	166	1.35	3.25
RT	1	295.2	279.1	225	1.24	3.42
GA-Basic	0.45	273.22	186.97	246	0.18	1.88

7. Conclusion

In this paper, we proposed four new heuristics for search-based, black-box automated testing of Real-Time Embedded Systems (RTES) based on a model of their environment. The heuristics were developed to exploit various properties of these environment models in an attempt to reach environments states indicating a fault in the RTES (Error states). We provide an extensive empirical evaluation on an industrial case study and thirteen artificial RTES that we developed based on two industrial case studies belonging to different domains. The models of these artificial problems present varying properties that may affect the performance of these heuristics and are meant to help us understand the conditions under which they are beneficial. We evaluated the individual heuristics and their 16 combinations with two search algorithms, Genetic Algorithms (GA) and (1+1) Evolutionary Algorithm (EA). We also used Random Testing (RT) as a comparison baseline.

Results show that when reaching a state adjacent to the error state (risky state) is not trivial (i.e., reached by random test cases), RT is significantly worse than any of the proposed search algorithms. In this case, the best results are obtained when using (1) a heuristic favoring test cases maximizing the time spent in risky states and (2) (1+1) EA as a search algorithm, which showed to be overall superior to GA. However, the heuristic that favored higher coverage of states in the environment model (coverage) showed significantly poorer performance with (1+1) EA in four of the thirteen problems. Based on the results, we proposed a way to combine RT with (1+1) EA in order to achieve high fault detection rates in practice.

Acknowledgments

The work presented in this paper was supported by the Norwegian Research Council and was produced as part of the ITEA 2 VERDE project. We are thankful to our industrial partners at Tomra and WesternGeco for their support throughout the project.

8. References

- [1] A. Arcuri, M. Iqbal, and L. Briand, "Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing," in *Testing Software and Systems*. Springer Berlin / Heidelberg, 2010, pp. 95-110.
- [2] M. Z. Iqbal, A. Arcuri, and L. Briand, "Automated System Testing of Real-Time Embedded Systems Based on Environment Models," Simula Research Laboratory, Technical Report (2011-19) 2011.
- [3] OMG, "Unified Modeling Language Superstructure, Version 2.3," <http://www.omg.org/spec/UML/2.3/>, ed, 2010.
- [4] OMG, "Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0," <http://www.omg.org/spec/MARTE/1.0/>, ed, 2009.
- [5] M. Z. Iqbal, A. Arcuri, and L. Briand, "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies," in *Model Driven Engineering Languages and Systems*. Springer Berlin / Heidelberg, 2010, pp. 286-300.
- [6] M. Z. Iqbal, A. Arcuri, and L. Briand, "Code Generation from UML/MARTE/OCL Environment Models to Support Automated System Testing of Real-Time Embedded Software," Simula Research Laboratory, Technical Report (2011-04) 2011.
- [7] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random Testing: Theoretical Results and Practical Implications," *IEEE Transactions on Software Engineering*, vol. 38, pp. 258-277, 2012.
- [8] M. Harman, S. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," Department of Computer Science, King's College London, TR-09-032009.
- [9] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation " *IEEE Transactions on Software Engineering*, vol. 36, pp. 742-762, 2010.
- [10] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability*, doi: 10.1002/stvr.457, 2011.
- [11] B. M. Broekman and E. Notenboom, *Testing Embedded Software*: Addison-Wesley Co., Inc., 2003.
- [12] D. Clarke and I. Lee, "Testing real-time constraints in a process algebraic setting," in *Proceedings of the 17th International Conference on Software Engineering*, 1995, pp. 51-60.
- [13] M. Krichen and S. Tripakis, "Conformance testing for real-time systems," *Formal Methods in System Design*, vol. 34, pp. 238-304, 2009.
- [14] B. Nielsen and A. Skou, "Automated test generation from timed automata," *International Journal on Software Tools for Technology Transfer*, vol. 5, pp. 59-77, 2003.
- [15] T. Mucke and M. Huhn, "Generation of optimized testsuites for UML statecharts with time," in *Testing of Communicating Systems*. Springer Berlin / Heidelberg, 2004, p. 128.

- [16] M. Zheng, V. Alagar, and O. Ormandjieva, "Automated generation of test suites from formal specifications of real-time reactive systems," *The Journal of Systems & Software*, vol. 81, pp. 286-304, 2008.
- [17] M. Auguston, M. J. B, and M. Shing, "Environment behavior models for automation of testing and assessment of system safety," *Information and Software Technology*, vol. 48, pp. 971-980, 2006.
- [18] L. Briand, Y. Labiche, and M. Shousha, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines*, vol. 7, pp. 145-170, 2006.
- [19] P. McMinn, "Search based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, pp. 105-156, 2004.
- [20] M. Heisel, D. Hatebur, T. Santen, and D. Seifert, "Testing Against Requirements Using UML Environment Models," in *Fachgruppentreffen Requirements Engineering und Test, Analyse & Verifikation*, 2008, pp. 28-31.
- [21] N. Adjir, P. Saqui-Sannes, and K. M. Rahmouni, "Testing Real-Time Systems Using TINA," in *Testing of Software and Communication Systems*. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009.
- [22] K. G. Larsen, M. Mikucionis, and B. Nielsen, "Online Testing of Real-time Systems Using Uppaal," in *Formal Approaches to Software Testing*. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005.
- [23] J. Peleska, F. Lapschies, E. Vorobev, H. Loeding, P. Smuda, H. Schmid, and Z. C., "A real-world benchmark model for testing concurrent real-time systems in the automotive domain," in *Testing Software and Systems*. Springer Berlin Heidelberg, 2011, pp. 146-161.
- [24] R. Lefticaru and F. Ipate, "Functional search-based testing from state machines," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2008, pp. 525-528.
- [25] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation," presented at the 11th International Conference on Quality Software, 2011.
- [26] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*: John Wiley and Sons, 2001.
- [27] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, pp. 608-624, 2006.
- [28] A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering " in *International Symposium on Search Based Software Engineering (SSBSE)*, 2011.
- [29] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1 - 10
- [30] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, pp. 3-16, 2004.

Combining Search-based Testing and Adaptive Random Testing to Improve Environment Model-based Testing of Real-time Embedded Systems

Muhammad Zohaib Iqbal, Andrea Arcuri, Lionel Briand

Submitted to a conference, 2012

Abstract – Effective system testing of real-time embedded systems (RTES) requires a fully automated approach. One such black-box system testing approach is to use environment models to automatically generate test cases and test oracles along with an environment simulator to enable early testing of RTES. In this paper, we propose a hybrid strategy, which combines (1+1) Evolutionary Algorithm (EA) and Adaptive Random Testing (ART), to improve the overall performance of system testing that is obtained when using each single strategy in isolation. An empirical study is carried out on a number of artificial problems and one industrial case study. The novel strategy shows significant overall improvement in terms of fault detection compared to individual performances of both (1+1) EA and ART.

1. Introduction

Real-time embedded systems (RTES) are widely used in critical domains where high system dependability is required. These systems typically work in environments comprising of large numbers of interacting components. The interactions with the environment are typically bounded by time constraints. Missing these time deadlines, or missing them too often for soft real-time systems, can lead to serious failures leading to threats to human life or the environment. There is usually a great number and variety of stimuli from the RTES environment with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. Testing all possible sequences of stimuli is not feasible. Hence, systematic automated testing strategies that have high fault revealing power are essential for effective testing of industry scale RTES. The system testing of RTES requires interactions with the actual environment. Since, the

cost of testing in actual environments tends to be high, environment simulators are typically used for this purpose.

In our earlier work, we proposed an automated system testing approach for RTES software based on environment models [1, 2]. The models are developed according to a specific strategy using the Unified Modeling Language (UML) [3], the Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile [4] and our proposed profile [5]. These models of the environment were used to automatically generate an environment simulator [6], test cases, and obtain test oracle [1, 2].

In our context, a test case is a sequence of stimuli generated by the environment that is sent to the RTES. A test case can also include changes of state in the environment that can affect the RTES behavior. For example, with a certain probability, some hardware components might break, and that affects the expected and actual behavior of the RTES. A test case can contain information regarding when and in which order to trigger such changes. So, at a higher level, a test case in our context can be considered as a setting specifying the occurrence of all these environment events in the simulator. Explicit “error” states in the models represent states of the environment that are only reached when RTES is faulty. Error states act as the oracle of the test cases, i.e., a test case is successful in triggering a fault in the RTES if any of these error states is reached during testing.

In previous work, we investigated several testing strategies to generate test cases. We used random testing (RT) [7] as baseline, and then considered two different approaches: Search-based Testing (SBT) [8] and Adaptive Random Testing (ART) [1]. For SBT, an *order function* was defined that utilizes the information in environment models to guide the search toward the error states. In contrast, with ART, test cases are rewarded based on their *diversity*. The results indicated that, apart from the failure rate of the system under test (SUT), the effectiveness of a testing algorithm also depends on the characteristics of the environment models. For problems where the environment model is easier to cover or where the failure rate of the RTES is high, even RT outperforms SBT. However, for more complex problems, SBT showed much better performance than RT. This raised the need for a strategy that combines the individual benefits of the two strategies and utilizes adaptive mechanisms based on the feedback from executed test cases.

In this paper, we extend our previous work by devising such a hybrid strategy that aims at combining the best search technique, i.e., (1+1) Evolutionary Algorithm (EA) in our experiments and ART (which is the algorithm that gave best results in our earlier

experiments in [2]) in order to achieve better overall results in terms of fault detection. We defined two different strategies for combining these algorithms, but due to space constraints, in this paper, we only discuss the strategy that showed the best results. The hybrid strategy (HS) discussed here starts with running (1+1) EA and switches to ART when (1+1) EA stops yielding fitter test cases. The decision of when to switch (referred to as *configuration*) can have significant impact on the performance of the strategy and one main objective of this paper is to empirically investigate different configuration options. The other combination strategy started by running ART and later switched to (1+1) EA if consecutive test cases generated through ART showed better fitness compared to previously executed test cases. It did show improvements over the individual algorithms, but fared worse than HS.

We evaluate the fault detection effectiveness of HS by performing a series of experiments on 13 artificial problems and an industrial case study. The RTES of the artificial problems were based on the specifications of two industrial case studies. Their environment models were developed in a way to vary possible modeling characteristics so as to understand their effect on the performance of the test strategies. We could not have covered such variations in environment models with one or even a few industrial case studies, hence the motivation to develop artificial cases. The industrial case study used is of a marine seismic acquisition system, which was developed by a company leading in this industry sector. For all these cases, we compared the performance of HS (with best configuration) with that of ART, (1+1) EA, and RT. The results suggest that in terms of success rates (number of times an algorithm found a fault within a given test budget), for the problems where RT/ART showed better performance over (1+1) EA, HS results are similar to ART/RT and for the problems where (1+1) EA was better, HS results are similar to those of (1+1) EA, thus suggesting that HS combines the strength of both algorithms.

The rest of the paper is organized as follows. Section 2 discusses the related work, while Section 3 provides an introduction to the earlier proposed environment model-based system testing methodology that we improve in this paper. Section 4 describes the proposed hybrid strategy, whereas Section 5 reports on the empirical study carried out for evaluation purposes. Finally, Section 6 concludes the paper.

2. Related Work

Depending on the goals, testing of RTES can be performed at different levels: model-in-the-loop, hardware-in-the-loop, processor-in-the-loop, and software-in-the-loop [9]. Our approach falls in the software-in-the-loop testing category, in which the embedded software is tested on the development platform with a simulated environment. The only variation is that, rather than simulating the hardware platform, we use an adapter for the hardware platform that forwards the signals from the SUT to the simulated environment. This approach is especially helpful when the software is to be deployed on multiple hardware platforms or the target hardware platform is stable.

There are only a few works in literature that discuss RTES testing based on environment models rather than system models. Auguston *et al.* [10] discusses the modeling of environment behaviors for testing of RTES using an event grammar. The behavioral models contain details about the interactions with the SUT and possible hazardous situations in the environment. Heisel *et al.* [11] propose the use of a requirement model and an environment model along with the model of the SUT for testing. Adjir *et al.* [12] discuss a technique for testing RTES based on the system model and assumptions in the environment using Labeled Prioritized Timed Petri Nets. Larsen *et al.* [13] propose an approach for online RTES testing based on time automata to model the SUT and environmental constraints. Iqbal *et al.* [5] propose an environment modeling methodology based on UML and MARTE for black-box system testing. Fault detection effectiveness of testing strategies based on these models was evaluated and reported in [8], including RT/ART [1], GA, and (1+1) EA. The results indicate that SBT show significantly better performance over RT for a number of cases and significantly worse performance than RT for a number of other cases.

There has been some work to combine SBT with RT. Andrews *et al.* propose the use of GA to tune parameters for random unit testing [14]. An evolutionary ART algorithm that uses the ART distance function as a fitness function for GA is proposed in [15]. In [16], the authors propose a search-based ART algorithm by using a variant of ART distance function as the fitness function for Hill Climbing to optimize the results of ART when the input domains are more than two dimensional.

The work presented here improves the work on environment model-based testing presented in [8] by combining the strengths of both ART and (1+1) Evolutionary

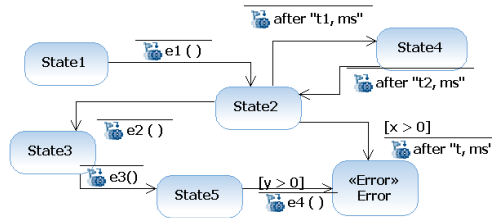


Figure. 1. A dummy state machine to explain search heuristics

Algorithm. Approaches discussed in the literature for combining ART/RT with SBT are restricted to improving ART or tuning RT by using search techniques. In contrast, here we want to use (1+1) EA to generate test cases that exploit the characteristics of environment models as well as benefit from the test diversity generated by ART, thus combining the two approaches.

3. Environment Model-based Testing

In this section, we discuss in more details the various components of our environment model-based testing approach.

3.1. Environment Modeling & Simulation

For RTES system testing, as we observed with our industry partner, software engineers familiar with the application domain would typically be responsible for developing the environment models. Therefore, we selected UML and its extensions as the environment modeling language, which is a standard modeling language that is widely taught and accepted by software engineers, in addition to be widely supported in terms of tools and training material. These are important considerations for successful industry adoption.

The environment models consist of a domain model and several behavioral models. The domain model, represented as a class diagram, captures the structural details of the RTES environment, such as the environment components, their relationships, and their characteristics. The behavior of the environment components is captured by state machines. These models are developed, based on our earlier proposed methodology by using UML, MARTE, and our proposed profile for environment modeling [5]. These models not only include the nominal functional behavior of the environment components (e.g., booting of a component) but also include their robustness (failure) behavior (e.g., break down of a sensor). The latter are modeled as “failure” states in the environment models. The behavioral models also capture what we call “error” states. These are the

states of the environment that should never be reached if the SUT is implemented correctly (e.g., no incorrect or untimely message from the SUT to the environment components). Therefore, error states act as oracles for the test cases.

An important feature of these environment models is that they capture the non-determinism in the environment, which is a common characteristic for most RTES environments. Non-determinism may include, for example, different occurrence rates and patterns of signals, failures of components, or user commands. Each environment component can have a number of non-deterministic choices whose exact values are selected at the time of testing. Java is used as an action language and OCL (Object Constraint Language) is used to specify constraints and guards.

Using model to text transformations, the environment models are automatically transformed into environment simulators implemented in Java. The transformations follow specific rules that we discussed in detail in [6]. During simulation a number of instances can be created for each environment component, which can interact with each others and the SUT (for example multiple instances of a sensor component). The generated simulators communicate with the SUT through a communication layer (e.g., TCP layer), which is written by software engineers. They are also linked with the test framework that provides the appropriate values for each simulation execution. The choice of Java as target language is based on actual requirements of our industrial partner, where the RTES under study only involves soft real-time constraints.

3.2. Testing RTES based on Environment Models

In our context, a test case execution is akin to executing the environment simulator. During the simulation, values are required for the non-deterministic choices in the environment models. A test case, in our context, can be seen as a test data matrix, where each row provides a series of values for a non-deterministic choice of the environment component (the number of rows is equal to the number of non-deterministic choices). Each time a non-deterministic choice needs to be made, a value from the corresponding matrix row is selected.

During the simulation, a query for a non-deterministic choice can be made several times and the number of queries cannot be determined before simulation. To resolve this problem, each matrix row (a data vector) is represented as a variable size vector so that whenever the end of a vector is reached, its size is increased at run time and new values are

added. In our earlier work [2], we evaluated the effect of the representations and starting lengths of the test data vectors on the fault detection effectiveness and showed that such a variable size vector is a suitable solution to this problem. In [1], we applied various testing strategies to generate test cases from the environment models, including ART, RT, and Genetic Algorithms (GA).

Given a test data matrix, a test case can be run for any arbitrary length of time (e.g., 10 seconds, one hour). The choice of the duration has high impact on the testing performance. Is it better to have many quick simulations, or fewer longer ones? This is conceptually similar to the choice of test length in test data generation of object-oriented software. In this paper, we choose a fixed duration based on the properties the models (e.g., if there are time transitions that take 10 seconds, then we should have test cases running for at least 10 seconds, otherwise those transitions will never be taken).

To calculate the distance between two test data matrices m_1 and m_2 for ART we use the function $dis(m_1, m_2) = \sum_r \sum_c abs(m_1[r,c] - m_2[r,c]) / |D(r)|$, where r and c represent the rows and columns of the matrices. In other words, we sum the absolute difference of each variable weighted by the cardinality of the domain of that variable. Often, these variables represent the time in timeout transitions. Therefore, ART rewards diversity for the values of non-deterministic choices. The results of the first experiments we conducted showed that RT/ART perform better than SBT [1].

For search-based testing, rather than using a fitness function, we use an *order* function. An order function is used to determine whether one solution is better than another, without having the problem of defining a precise numerical score (this is often difficult when several objectives need to be combined and tight budget constraints do not allow a full multi-objective approach). The new order function h can be seen as an extension of the fitness function developed for model-based testing based on system specifications [17]. The original fitness function uses the so-called “approach level” and normalized “branch distance” to evaluate the fitness of a test case. For environment model-based testing, we introduced the concept of “time distance” with a look-ahead branch distance and the concept of “time in risky states” [8].

In our context, the goal is to minimize the order function h , which heuristically evaluates how far a test case is from reaching an error state. If a test case with test data m is executed and an error state of the environment model is reached, then $h(m) = 0$. The approach level (A) refers to the minimum number of transitions in the state machine that

are required to reach the error state from the closest executed state. Figure. 1 shows a dummy example state machine to elaborate the concept. The state named *Error* is the error state. Events $e1$, $e2$, and $e3$ are signal events, whereas events *after* “ t , s ”, *after* “ $t1$, ms ”, and *after* “ $t2$, ms ” are time events with t , $t1$, and $t2$ as the time values and ms and s as time units referring to milliseconds and seconds. Events $e3$ and *after* “ t , s ” are guarded by constraints using OCL. If the desired state is *Error* and the closest executed state was *State5*, then the approach level is 1.

The approach level is helpful to reward test case executions that get closer to an error state, but it does not provide any gradient (guidance) to solve the possible guards on the state transitions. The branch distance (B) is used to heuristically score the evaluation of the guards on the outgoing transitions from the closest executed state. In previous work [18], we have defined a specific branch distance function for OCL expressions that is reused here for calculating the branch distance. In the dummy state machine in Figure. 1 we need to solve the guard “ $y > 0$ ” so that whenever $e4$ is triggered, then the simulation can transition to *Error*. Note that branch distance is less important than approach level, since it is required only when the transition towards an error state is guarded and the approach level cannot be reduced any further.

The third important part of the order function is the time distance (T), which comes into play when there are timeout transitions in the environment models. For example, in Figure. 1, the transition from *State2* to *Error* is a timeout transition. If a transition should be taken after z time units, but it is not, we calculate the maximum consecutive time c the component stayed in the source state of this transition (e.g., *State2* in the dummy example). To guide the search, we use the following heuristic: $T = z - c$, where $c \leq z$. For transitions other than time transitions, we initially decided to calculate branch distance after an event is triggered. As investigated in our earlier work [8], this is not suitable for time transitions and therefore the concept of a look-ahead branch distance (LB) was introduced. LB represents the branch distance of OCL guard on a time transition when it is not fired (i.e., the timeout did not occur). Because OCL evaluations are free from side-effects [18], this approach is feasible in our context.

The fourth important part of the order function is “time in risky states” (TIR). TIR favors the test cases that spent more time in the state adjacent to the error state (i.e., the *risky* state). The motivation behind this heuristic is that, the more time spent in a risky state, the higher the chances of events happening in the environment or SUT that lead to

the error state (e.g., receive a signal from the SUT). For example, for the state machine shown in Figure. 1, this heuristic will favor the test cases that spend more time in the risky states *State2* or *State5*. For instance in *State2*, it is possible to increase the value of *t1* in the time event *after* “*t1, ms*”, which will increase the time spent in this state. *TIR* is less important than the other three heuristics and is only used when the other heuristics fail to guide the search. The order function *h* using the four previously described heuristics, given two test data matrices m_1 and m_2 as input, is defined as:

$$\begin{aligned}
 h(m_1, m_2) &= \begin{cases} v(m_1, m_2) & \text{if } v(m_1, m_2) \neq 0 \\ \mathbf{1} & \text{if } v(m_1, m_2) = 0 \text{ and } TIR_{sum}(m_1) > TIR_{sum}(m_2) \\ \mathbf{0} & \text{if } v(m_1, m_2) = 0 \text{ and } TIR_{sum}(m_1) = TIR_{sum}(m_2) \\ \mathbf{-1} & \text{otherwise} \end{cases} \quad (1) \\
 v(m_1, m_2) &= \begin{cases} \mathbf{1} & \text{if } A_{min}(m_1) < A_{min}(m_2) \text{ or } (A_{min}(m_1) = A_{min}(m_2) \text{ and } B_{min}(m_1) < \\ & B_{min}(m_2)) \text{ or } (A_{min}(m_1) = A_{min}(m_2) \text{ and } B_{min}(m_1) = B_{min}(m_2) \\ & \text{and } ITD_{min}(m_1, m_2) = 1) \\ \mathbf{0} & \text{if } A_{min}(m_1) = A_{min}(m_2) \text{ and } B_{min}(m_1) = B_{min}(m_2) \text{ and } \\ & ITD_{min}(m_1, m_2) = 0 \\ \mathbf{-1} & \text{otherwise} \end{cases} \\
 ITD_e(m_1, m_2) &= \begin{cases} \mathbf{1} & \text{if } LB_e(m_1) < LB_e(m_2) \text{ or } (LB_e(m_1) = LB_e(m_2) \text{ and } T_e(m_1) < \\ & T_e(m_2)) \\ \mathbf{0} & \text{if } (LB_e(m_1) = LB_e(m_2) \text{ and } T_e(m_1) = T_e(m_2)) \\ \mathbf{-1} & \text{otherwise} \end{cases}
 \end{aligned}$$

where for an error state e , $A_{min}(m)$ represents the minimum approach level over all error states, $B_{min}(m)$ represents the minimum branch distance, T_e represents the time distance, LB_e is the look-ahead branch distance for an error state e , and $TIR_{sum}(m)$ is the sum of time spent in risky states for all error states for test data matrix m .

The results, based on our extensive experiments evaluating various heuristics [8], suggested that (1+1) EA with the order function in (1) gave best results in cases where the approach to a risky state was non-trivial (i.e., simulation cannot reach a risky state in <5 random test cases). But in cases where the approach was easy, RT outperformed evolutionary algorithms.

4. Hybrid Strategy by Combining Adaptive Random and Search-based Testing

In this section we present our proposed hybrid strategy (HS) that combines (1+1) EA and ART to improve the overall fault detection effectiveness of our system testing approach. As discussed earlier (Section 3), previous studies showed that, in some cases, RT/ART could perform better than SBT. The difference between their performances was mostly significant and at times even extreme. In [2] and [8], we identified two possible reasons for this behavior. First of all, for the problems with high failure rates, randomized algorithms were found to be much better than SBT [2]. For high failure rates, there is no need for search, as solutions are anyway found quickly. Crossover produces similar genes, while mutation only performs small modifications. This can have a negative effect as, given just few fitness evaluations, only similar solutions are evaluated (in contrast to RT/ART). Secondly, the performance of the algorithms also depended on the properties of environment models, and in particular how easy is it to traverse the models in order to reach the error states. In other words, by combining ART and (1+1) EA, we hope to achieve a consistently good result regardless of the properties of the SUT or its environment.

In the environment models, there are transitions on paths leading to error states that depend only on the behavior of the SUT (i.e., they can only be triggered when the SUT behaves in a certain way). Transition from a risky state to an error state is one such example as it is only triggered when the SUT behaves in an erroneous way. Another example can be when a guard on a transition depends on a specific response from the SUT. To execute this behavior of SUT, the overall environment (combination of environment components) needs to behave in a particular way. This particular behavior of the environment that is required to trigger SUT behavior cannot be determined before simulation, since for practical reasons discussed earlier the design of the SUT is not visible. Hence, the information of what should be executed in the environment to trigger this behavior is not available in the environment models. The fitness function for SBT (which exploits the environment models to guide the search towards error states) in this case does not give enough gradient to generate fitter test cases (i.e., a search plateau). In these cases maximizing the diversity of the environment behavior (e.g., by using entirely different values for the test data matrix, irrespective of their effect on the fitness) appears

to be a better option, thus favoring RT/ART. This can explain the scenarios where RT/ART show better results than (1+1) EA.

On the other hand, if in the environment models, there are transitions on the path to error states which are triggered by specific behaviors of the environment (e.g., a transition triggered as a result of a specific non-deterministic event in the environment, such as a failure of an environment component) or time transitions, then fitness function for SBT is specifically designed to deal with these cases and are more suitable for such cases than RT. For example, in the fitness function, the time distance heuristic is defined specifically for time transitions and favors test cases that are closer to executing the transitions (i.e., with a value of c closer to z , see Section 3.2). OCL constraints in guards that are independent of SUT behavior but dependent on the state of environment components (e.g., a constraint requiring a sensor to be broken), can be solved by directly changing the values of these components' attributes. For such constraints, our previous results showed that SBT are an order of magnitude better than RT [18].

HS combines ART, which showed best results in our initial experiments [2], with our proposed SBT strategy that showed best performance [8], i.e., (1+1) EA with improved time distance and the “time in risky state” heuristic (ITD-TIR). The strategy is designed to combine the strengths of both (1+1) EA and ART. This strategy starts by applying (1+1) EA. If (1+1) EA does not find fitter test cases after running n number of test cases, the testing algorithm is switched to ART. All the test cases that were executed so far are now used for distance calculations in ART. Figure. 2 shows the pseudo-code for HS. The idea behind switching from (1+1) EA to ART is that there is not enough time for a random walk to get out of a fitness plateau. And so, in this scenario, applying ART can yield better results. Running system test cases is very time consuming, so only few fitness evaluations are feasible within reasonable time (e.g., 1000 test cases can already take several hours). Therefore, in case of fitness plateau, it is reasonable to switch strategy, and rather reward diversity instead of the fitness value. Though the choice of n is arbitrary it can have significant consequences on the performance of this strategy. A too small value of n will result in an early switch to ART. If the given problem matches the case where (1+1) EA performs better, then the performance of HS will be affected. Similarly, if n is too large then the remaining testing budget might not be sufficient for ART to perform well.

Algorithm HybridStrategy(mx, n, w)

Input mx : number of maximum fitness evaluations
 n : number of consecutive test cases with no improved fitness
 w : number of random test-cases to generate for comparison in ART

Declare Y : set of executed test cases = {}, W : set of randomly generated test cases = {}
 ev : number of fitness evaluations performed = 0
 z : number of consecutive test cases with no improved fitness found so far = 0
 T_c : a random test case, T_m : mutated test case, T_w : a test case from W , T_e test case from W selected according to ART criteria
 D_w : minimum distance of test case T_w with all the test cases in Y
 d : stores the maximum value of D_w obtained over W

1. **begin**
2. Generate a random test case T_c
3. Execute T_c and evaluate whether environment error state is reached
4. Add T_c to Y
5. **while** environment error state not reached *OR* $ev \leq mx$ *OR* $z \leq n$
6. Mutate T_c to get T_m
7. Execute T_m and evaluate whether environment error state is reached
8. Add T_m to Y
9. Increment ev
10. **if** $fitness(T_m) \geq fitness(T_c)$
11. **then** $T_c = T_m, z = 1$
12. **else**
13. Increment z
14. **while** environment error state not reached *OR* $ev \leq mx$
15. Sample w random test cases and add them to W
16. $d = 0$
17. **for each** $T_w \in W$
18. Calculate D_w
19. **if** $D_w > d$
20. **then** $d = D_w, T_e = T_w$
21. Execute T_e and evaluate whether environment error state is reached
22. Add T_e to Y
23. Increment ev
24. **end**

Figure. 2 Pseudo code of the proposed hybrid strategy (HS)

5. Empirical Study

The objective of this empirical study is to evaluate the fault detection effectiveness of the proposed hybrid strategy.

5.1. Case Study

To enable experimentation with diverse environment models and RTES, we developed 13 different artificial RTES that were inspired by two industrial cases we have been involved with [2] and one case study discussed in the literature [19]. Since, there are no benchmark

RTES available to researchers, we specifically designed these artificial problems to conduct our experiments (called P1 – P13). The goal while developing the models of these RTES was to vary various characteristics of the environment models (e.g., guarded time transitions, loops) that were expected to have an impact on the test heuristics. Nine of these artificial problems were inspired by a marine seismic acquisition system developed by one of our industrial partners. They covered various subsets of the environment of that RTES. Three problems were inspired by the behavior of another industrial RTES (automated recycling machine) developed by another industrial partner. The thirteenth artificial problem was inspired by the train control gate system described in [19].

These RTES are multithreaded, written in Java and they communicate with their environments through TCP. Each of the artificial problems had one error state in their environment models and non-trivial faults were introduced by hand in each of them. We could have rather seeded the faults in a systematic way, as for example by using a mutation testing tool [20] but opted for a different procedure since the SUTs are highly multithreaded and use a high number of network features (e.g., opening and reading/writing from TCP sockets), features that are not handled by current mutation testing tools. Furthermore, our testing is taking place at the system level, and though small modifications made by a mutation testing tool might be representative of faults at the unit level, it is unlikely to be the case at the system level for RTES. On the other hand, the faults that we manually seeded came from our experience with the industrial RTES and from the feedback of our industry partners.

The industrial case study we also report on (called IC) is a very large and complex seismic acquisition system (mentioned above) that interacts with many sensors and actuators. The timing deadlines on the environment are in the order of hundreds of milliseconds. The company that provided the system is a market leader in its field. For confidentiality reasons we cannot provide full details of the system. The SUT consists of two processes running in parallel, requiring a high performance, dedicated machine to run. For the industrial case study, we did not seed any fault and the goal was to find the real fault that we uncovered earlier [1].

5.2. Experiment

In this paper, we want to answer the following research questions:

RQ1. Which configuration is best in terms of fault detection for the proposed hybrid strategy (HS)?

RQ2. How the fault detection of the best HS configuration compares with the performance of ART, (1+1) EA, and RT for (a) the artificial problems (P1-13) and (b) the industrial case study (IC)?

To answer these research questions, we have conducted two distinct sets of experiments, one for the artificial problems (to answer RQ1 and RQ2a) and one for the industrial RTES (to answer RQ2b). For test case representation in these experiments we used a dynamic representation with a length equal to 10 for the test cases (which correspond to each row of the test data matrix m). In our earlier experiments this setting showed the best results [2]. For (1+1) EA we calculated the mutation rate as $1/k$, where k is the number of total elements in a test data matrix. This strategy is widely used for SBT and was initially suggested in [21]. We used the fitness function that performed best in our previous experiments [8], as discussed in Section 4: Improved Time Distance with Time in Risky State (ITD-TIR).

To answer RQ1, we used 12 different values for the number of test cases which fitness should be considered before switching from (1+1) EA to ART: $n \in \{10, 20, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500\}$. We ran these 12 configurations on each of the 13 artificial problems. To answer RQ2a, we selected the configuration of HS that gave the best result in terms of fault detection for the 13 artificial problems. We compared this configuration with the results of (1+1) EA, ART, and RT on these problems. RT was used as a comparison baseline.

For the artificial problems, the execution time of each test case was fixed to 10 seconds and we stopped each algorithm after 1000 sampled test cases or as soon as we reached any of the error states. The choice of running each test case for 10 seconds was based on the properties of the RTES and the environment models. The objective was to allow enough time for the test cases to reach an error state. We ran each of the strategies 20 times on each artificial problem with different random seeds. The total number of sampled test cases was 1,561,390, which required around 180 days of CPU resources. Therefore, we performed these experiments on a cluster of computers.

To answer RQ2b, we carried out experiments on the described seismic acquisition system. We run each test case for 60 seconds, where 1000 test case executions (fitness

evaluations) can take more than 16 hours. This choice has been made based on the properties of the RTES and discussions with the actual testers. Due to the large amount of resources required, we only ran the configuration that on average gave best results for the artificial problems (i.e., $n=50$) and compared its fault detection rate with that of (1+1) EA, ART, and RT. We carried out 39 runs for each of these four test strategies. The total number of sampled test cases was 55,283, which required over 55 days of computation on a single, high-performance, dedicated machine.

Table 1. Success Rates (SR) for 12 configurations of HS on the 13 problems

Configurations → Problems ↓	10	20	50	60	70	80	90	100	200	300	400	500
P1	0.5	0.75	0.95	1	1	1	1	1	1	1	1	1
P2	0.85	0.95	1	1	1	1	1	1	0.9	1	1	1
P3	1	1	1	1	1	1	1	1	0.9	0.8	0.6	0.5
P4	0.05	0.2	0.8	0.85	0.7	0.75	0.9	0.9	1	1	0.9	1
P5	0.85	1	1	1	1	1	1	1	1	1	1	1
P6	0	0.15	0.45	0.4	0.45	0.5	0.45	0.6	0.7	0.7	0.5	0.6
P7	0.3	0.4	0.8	0.8	0.85	0.95	0.8	0.8	0.8	0.8	0.8	1
P8	1	1	1	1	1	1	1	1	1	1	0.95	1
P9	0.05	0.05	0.45	0.55	0.55	0.35	0.6	0.4	0.8	0.45	0.5	0.55
P10	1	1	1	1	0.95	0.85	1	0.95	0.65	0.55	0.4	0.45
P11	1	1	1	0.95	0.95	0.9	1	0.9	0.65	0.05	0.1	0.4
P12	1	1	1	1	0.95	1	1	1	0.9	0.9	0.75	0.65
P13	1	1	1	1	1	1	1	1	0.9	0.7	0.95	0.85
Average SR	0.66	0.73	0.88	0.89	0.88	0.87	0.9	0.89	0.86	0.77	0.73	0.77
Average Rank	6.38	6.73	5.19	5.77	5.23	6.31	6.50	6.19	6.73	8.46	7.73	6.69

To analyze the results, we used the guidelines described in [22] which recommends a number of statistical procedures to assess randomized algorithms. First we calculated the success rates of each algorithm: the number of times it was successful in reaching the error state out of the total number of runs. These success rates are then compared using the Fisher Exact test, quantifying the effect size using an odds ratio (ψ) with a 0.5 correction. When the differences between the success rates of two algorithms were not significant, we then looked at the average number of test cases that each of the algorithms executed to reach the error state. We used the Mann-Whitney U-test and quantified the effect size with the Vargha-Delaney A_{12} statistics. The significance level for these statistical tests was set to 0.05.

Table 2. Success Rates of HS (Best configuration), RT, ART, and (1+1) EA

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	Avg.	IC
HS	0.95	1	1	0.8	1	0.45	0.8	1	0.45	1	1	1	1	0.88	1
ART	0.4	0.75	1	0	0.95	0	0.15	1	0	1	1	1	1	0.63	1
EA	1	1	0.5	1	1	0.7	0.85	1	0.35	0.45	0	0.7	0.95	0.73	0.74
RT	0.45	1	1	0	0.65	0	0.2	1	0	1	1	1	1	0.64	0.97

Table 3. Comparison of best HS configuration with RT, ART, & (1+1)EA*

Problem	HS vs. (1+1) EA	HS vs. RT	HS vs. ART
P1	-	p = 0.0012, $\psi = 15.74$	p = 0.0004, $\psi = 19.12$
P2	-	it-p = 0.0065, $A_{12} = 0.25$	p = 0.047, $\psi = 14.55$
P3	p = 0.0004, $\psi = 41.00$	-	<i>it-p = 0.013, $A_{12} = 0.73$</i>
P4	-	p = 1.5e-07, $\psi = 150.33$	p = 1.5e-07, $\psi = 150.33$
P5	-	p = 0.0083, $\psi = 22.78$	-
P6	-	p = 0.0012, $\psi = 33.87$	p = 0.0012, $\psi = 33.87$
P7	-	p = 0.0004, $\psi = 13.44$	p = 8.7e-05, $\psi = 18.33$
P8	-	<i>it-p = 0.009, $A_{12} = 0.74$</i>	<i>it-p = 0.0004, $A_{12} = 0.825$</i>
P9	-	p = 0.0012, $\psi = 33.87$	p = 0.0012, $\psi = 33.87$
P10	p = 0.0001, $\psi = 49.63$	<i>it-p = 0.0006, $A_{12} = 0.81$</i>	<i>it-p = 0.0002, $A_{12} = 0.85$</i>
P11	p = 1.4e-11, $\psi = 1681.00$	-	<i>it-p = 0.0032, $A_{12} = 0.77$</i>
P12	p = 0.02, $\psi = 18.38$	<i>it-p = 0.0016, $A_{12} = 0.79$</i>	<i>it-p = 0.0008, $A_{12} = 0.81$</i>
P13	-	<i>it-p = 0.0199, $A_{12} = 0.71$</i>	<i>it-p = 0.021, $A_{12} = 0.71$</i>
IC	p = 0.0004, $\psi = 28.83$	-	<i>it-p = 0.015, $A_{12} = 0.66$</i>

5.3. Results & Discussion

Table 1 provides the success rates (in terms of fault detection) for various HS configurations. The last row of the table shows the average ranking of each configuration based on the statistical differences among them. Configurations that are statistically equivalent (i.e., p-values above 0.05) are assigned a similar ranking. This is done by assigning scores based on pairwise comparisons of configurations. Whenever a configuration is better than the other and the difference is statistically significant, its score is increased (for details, see [22]). Then, based on the final scores, each configuration is assigned ranks ranging from 1 (best configuration) to 12 (worst configuration). In case of ties, ranks are averaged. As the success rates and average rankings indicate, using a very low (< 50) or very high value (≥ 200) of n results in a degraded performance for HS. With a low value of n , HS makes the switch from (1+1) EA to ART too early, which does not give sufficient time for (1+1) EA to converge and hence running HS becomes similar to only running ART. In cases where ART performs well, such configurations of HS also perform well (see Table 2 for the performance of ART on artificial problems). For

instance, for $n = 10$, the average success rate is 66% and average ranking is 6.38. Similarly, when HS switches too late, it does not give enough time to ART (given the upper bound of 1000 iterations) and hence running HS is similar to running (1+1) EA in such cases. These configurations perform well in cases where (1+1) EA performs well (Table 2) and poor otherwise. The best results are provided for values between 50 and 100 and the differences in results in this range are not significant. Though the results are not fully consistent across all problems, configuration $n = 50$ has the best average rank across all problems and is always very close to the maximum success rates. We can hence answer RQ1 by stating that, overall, $n=50$ shows the best results for HS and therefore this configuration can be used when applying HS on new problems.

For RQ2a we compared the best HS configuration ($n = 50$) with RT, ART, and (1+1)EA. Table 2 shows the corresponding success rates of these algorithms and Table 3 shows a comparison of HS with the other three algorithms based on statistical tests. The statistics for the situations where HS is significantly better are bold-faced and are italicized where it is significantly worse. Table cells with a '-' denote no significant differences. P-values obtained as a result of Fisher Exact test on the success rates are denoted as p and odds ratio as ψ . In cases where there is no statistical difference in success rates, the number of iterations is considered and the p-values of the Mann-Whitney U-test are denoted as $it-p$ and corresponding effect sizes by A_{12} .

When compared to (1+1) EA, HS showed better fault detection performance in four of the artificial problems (P3, P10 – P12) and had similar results otherwise. These are the problems where (1+1) EA, when ran in isolation, showed poor results when compared to RT and ART (as visible from Table 2). For example in the case of P11, (1+1) EA was not able to find the a in any of the runs. On the other hand it is 100% for HS, RT, and ART, which means that these strategies were able to find a fault in every run. Hence, HS shows significant improvement over (1+1) EA.

When compared to RT, HS showed significantly better results in terms of success rates for six artificial problems (P1, P4, P5, P6, P7, and P9) and had similar results for all the other problems. Similarly with ART, in terms of success rates, HS showed better results for six artificial problems (P1, P2, P4, P6, P7, and P9) and had similar results for the rest. P1, P4, P6, P7, and P9 are the problems where ART and RT showed poor results when compared to (1+1) EA (Table 2). For example in the cases of P4, P6, and P9, the success rate of both RT and ART is 0, but that of (1+1) EA and HS is 1 and 0.8, respectively. Hence, in terms

of success rates, HS shows significantly better results when compared to RT and ART. However, in terms of number of iterations required to detect the fault, HS is significantly worse than RT in four problems (P8, P10, P12, and P13) and significantly worse than ART in six problems (P3, P8, P10, P11, P12, and P13). But, for all these problems, the success rate of HS, RT, and ART is 1, which means that whenever these algorithms run they find the fault (within the budget of 1000 test cases). Therefore, we can answer RQ2a by stating that HS shows overall significantly better performance than ART, RT, and (1+1) EA in terms of fault detection, but was slower than RT/ART in finding faults for problems where these two algorithms perform better than (1+1) EA. But since the success rate of HS is 100%, and therefore the first run is expected to reach the error state, this difference in execution time has limited practical impact.

For RQ2b we compared the performance of the best configuration of HS ($n = 50$) with that of ART, RT, and (1+1) EA on the industrial case study. The last row of Table 3 shows a comparison of the results of the four strategies on this case study (IC) and the last column of Table 2 shows the corresponding success rates. The results are similar to that obtained for those artificial problems where RT and ART perform better than (1+1) EA. HS outperformed (1+1) EA. When compared with the results of ART and RT, there is no significant difference though (100% success rate). These results are consistent with RQ2a and we can therefore answer RQ2 by stating that, overall, HS shows significantly better results when compared to (1+1) EA, RT, and ART. However, as for RQ2a, for problems where ART performed much better than (1+1) EA, though the success rates of HS and ART are similar, ART find the faults faster than HS.

HS starts with (1+1) EA and switches only when *fifty* consecutive test cases do not show better fitness. Fitness evaluations make HS slower than ART/RT but its effectiveness considerably improves over ART/RT for the problems where they showed poor results. In the light of these results, we can conclude that when applying our testing approach, using HS seems to be the most practical choice as its performance, unlike that of (1+1) EA, ART, and RT, is not drastically affected by the properties of the SUT and its environment models. As a result, testers can apply this strategy in confidence, knowing it will perform well in most circumstances.

5.4. Threats to Validity

Although the artificial problems that we developed were based on industrial RTES and are not trivial to test (they are multithreaded and hundreds of lines long), these artificial problems may not be representative of complex RTES. To reduce this threat, we used artificial problems inspired by actual RTES and intentionally varied the properties of their environments in ways that could affect the testing strategies.

A typical problem when testing RTES is the accurate simulation of time. Our approach focuses on RTES with soft time deadlines in the order of hundreds of milliseconds with an acceptable jitter of a few milliseconds. Therefore, we used the CPU clock to represent time. This might be unreliable if time constraints in the RTES were very tight (e.g., nanoseconds) since they could be violated due to unpredictable changes of load balance in the CPU in the presence of unrelated process executions. To be on the safe side, to evaluate whether our results are reliable, we selected a set of experiments and ran them again with exactly the same random seeds. We obtained equivalent results with a small variance of a few milliseconds, which in our context did not affect the testing results.

6. Conclusion

In this paper, we proposed a hybrid strategy (HS) that combines (1+1) Evolutionary Algorithm (EA) and Adaptive Random Testing (ART) for black-box automated system testing of real-time embedded systems (RTES). The strategy was developed to combine the benefits of both algorithms, since their individual results varied greatly depending on the failure rate of the system under test and properties of its environment. The ultimate goal was to obtain a strategy with consistently good results. The proposed strategy starts with running (1+1) EA and switches to ART when the (1+1) EA search stops yielding fitter test cases. We empirically investigated when to switch to ART and identified an optimal setting for HS. Results indicate that switching too early or too late than the identified setting has a negative impact on the performance of the strategy. Based on the experiments, when using HS in practice, we propose switching to ART after (1+1) EA generates 50 consecutive test cases that do not improve fitness. We evaluated the proposed strategy and compared its performance with that of running (1+1) EA and ART individually. We also use random testing (RT) as a comparison baseline. The empirical evaluation uses an industrial case study and 13 artificial problems that were developed based on two industrial

case studies belonging to different domains. The models of these artificial problems were developed in order to vary their characteristics, thus potentially affecting the performance of the evaluated testing strategies. Overall, the results indicate that HS shows significantly better performance in terms of fault detection (an overall 88% success rate for artificial problems and 100% for the industrial case study) than the other three algorithms (for artificial problems: ART: 63%, RT: 64%, and (1+1) EA: 74% and for the industrial case study: ART: 100%, RT, 97%, (1+1) EA: 74%). Unlike the other strategies, variations in environment properties do not have a drastic impact on the performance of HS and it is therefore the most practical approach, showing consistently good results for different problems.

7. References

- [1] A. Arcuri, M. Iqbal, and L. Briand, "Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing," in *Testing Software and Systems*. Springer Berlin / Heidelberg, 2010, pp. 95-110.
- [2] M. Z. Iqbal, A. Arcuri, and L. Briand, "Automated System Testing of Real-Time Embedded Systems Based on Environment Models," Simula Research Laboratory, Technical Report (2011-19) 2011.
- [3] OMG, "Unified Modeling Language Superstructure, Version 2.3, <http://www.omg.org/spec/UML/2.3/>," ed, 2010.
- [4] OMG, "Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0, <http://www.omg.org/spec/MARTE/1.0/>," ed, 2009.
- [5] M. Z. Iqbal, A. Arcuri, and L. Briand, "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies," in *Model Driven Engineering Languages and Systems*. Springer Berlin / Heidelberg, 2010, pp. 286-300.
- [6] M. Z. Iqbal, A. Arcuri, and L. Briand, "Code Generation from UML/MARTE/OCL Environment Models to Support Automated System Testing of Real-Time Embedded Software," Simula Research Laboratory, Technical Report (2011-04) 2011.
- [7] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random Testing: Theoretical Results and Practical Implications," *IEEE Transactions on Software Engineering*, vol. 38, pp. 258-277, 2012.
- [8] M. Z. Iqbal, A. Arcuri, and L. Briand, "Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software " in *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.
- [9] B. M. Broekman and E. Notenboom, *Testing Embedded Software*: Addison-Wesley Co., Inc., 2003.
- [10] M. Auguston, M. J. B, and M. Shing, "Environment behavior models for automation of testing and assessment of system safety," *Information and Software Technology*, vol. 48, pp. 971-980, 2006.

- [11] M. Heisel, D. Hatebur, T. Santen, and D. Seifert, "Testing Against Requirements Using UML Environment Models," in *Fachgruppentreffen Requirements Engineering und Test, Analyse & Verifikation*, 2008, pp. 28-31.
- [12] N. Adjir, P. Saqui-Sannes, and K. M. Rahmouni, "Testing Real-Time Systems Using TINA," in *Testing of Software and Communication Systems*. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009.
- [13] K. G. Larsen, M. Mikucionis, and B. Nielsen, "Online Testing of Real-time Systems Using Uppaal," in *Formal Approaches to Software Testing*. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005.
- [14] J. H. Andrews, T. Menzies, and F. C. H. Li, "Genetic algorithms for randomized unit testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 80-94, 2011.
- [15] A. F. Tappenden and J. Miller, "A novel evolutionary approach for adaptive random testing," *IEEE Transactions on Reliability*, vol. 58, pp. 619-633, 2009.
- [16] C. Schneckenburger and F. Schweiggert, "Investigating the dimensionality problem of Adaptive Random Testing incorporating a local search technique," in *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW '08)*, 2008, pp. 241-250.
- [17] R. Lefticaru and F. Ipate, "Functional search-based testing from state machines," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2008, pp. 525-528.
- [18] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation," presented at the 11th International Conference on Quality Software, 2011.
- [19] M. Zheng, V. Alagar, and O. Ormandjieva, "Automated generation of test suites from formal specifications of real-time reactive systems," *The Journal of Systems & Software*, vol. 81, pp. 286-304, 2008.
- [20] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, pp. 608-624, 2006.
- [21] H. Mühlenbein, "How genetic algorithms really work: I. mutation and hillclimbing," *Parallel problem solving from nature*, vol. 2, pp. 15-25, 1992.
- [22] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1 - 10

Experiences of Applying UML/MARTE on Three Industrial Projects

Muhammad Zohaib Iqbal, Shaukat Ali, Tao Yue, Lionel Briand

Submitted to a conference, 2012

Abstract – MARTE (Modeling and Analysis of Real-Time and Embedded Systems) is a UML profile, which has been developed to model concepts specific to Real-Time and Embedded Systems (RTES). In previous years, we have we have applied UML/MARTE to three distinct industrial problems in various industry sectors: architecture modeling and configuration of large-scale and highly configurable integrated control systems, model-based robustness testing of communication-intensive systems, and model-based environment simulator generation of large-scale RTES for testing. In this paper, we report on our experiences of solving these problems by applying UML/MARTE on four industrial case studies. Based on our common experiences, we derive a framework to help practitioners for future applications of UML/MARTE. The framework provides a set of detailed guidelines on how to apply MARTE in industrial contexts and will help reduce the gap between the modeling standards and industrial needs.

Keywords: UML, MARTE, Real-time Embedded Systems, Architecture Modeling, Model-based Testing

1. Introduction

Model Based Engineering (MBE) consists in using models as the primary artifacts in various development phases of software systems, including, for example, configuration and software testing. The Unified Modeling Language (UML) [1] and its extensions (via its profiling mechanism) are the most widely used modeling notations for software systems in diverse domains.

Real-time embedded systems (RTES) are widely used in many different domains, as for example from integrated control systems to consumer electronics. Already 98% of computing devices are embedded in nature and it is estimated that, by the year 2020, there will be over 40 billion embedded computing devices worldwide [2]. Modeling for such

systems requires constructs that deal with characteristics specific to RTES (such as resource modeling, timeliness, schedulability). The recent UML profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) [3] is an effort to address the growing modeling needs of RTES.

In software engineering, like any engineering discipline, the usefulness of a new concept must ultimately be evaluated by applying it in real-life scenarios. To successfully apply MBE in practice, selecting a modeling language is not sufficient; rather we need to provide a detailed methodology on how to use the selected notations, which is a piece of information usually missing from language specifications and varies from problem to problem. This paper reports the experiences of four such applications on industrial RTES and based on the experiences layouts the guidelines that can be used for future successful application of UML/MARTE for RTES.

There are very few works discussing the experiences of using UML/MARTE. Demathieu *et al* [4] discuss their experiences of applying UML and MARTE on an academic case study for software resource modeling, hardware resource modeling, and modeling for logical system decomposition. Briand *et al* [5] discuss their experiences of applying MBE to three industrial cases belonging to maritime and energy domains using UML and MARTE. The work focuses on providing guidelines to improve collaboration between industry and researchers. Yue *et al* [6] discuss their experience of conducting a systematic and industrial domain analysis and the feasibility of applying model-based product line engineering methods for architecture modeling and configuration of large-scale integrated control systems. Espinoza *et al* [7] evaluate MARTE after applying it to a project in the automobile domain. Middleton *et al* [8] present their experiences about applying UML and MARTE for stochastic modeling of two interactive applications.

Our work discusses experiences of applying UML/MARTE on four industrial RTES belonging to different domains. We report our experiences of solving three industrial problems over the span of four years. The first problem was about architecture modeling and configuration of large-scale and highly configurable integrated control systems for FMC[9] Subsea Production Systems. The second problem was of model-based robustness testing of a video conferencing system at Cisco Systems [10]. The third problem was of environment model-based testing for a marine seismic acquisition system at WesternGeco [11] and an automated bottle recycling system at Tomra [12]. Based on our common experiences in the projects, we derived a comprehensive framework to successfully use

MARTE in future industrial applications. The framework, which is the first of its kind, aims at providing detailed guidelines and steps on how to apply and extend UML/MARTE in industrial contexts. .

The rest of the paper is organized as follows. Section 2 provides the background, while Section 3 discusses the contexts, modeling solutions and key results for the four selected industrial problems. Section 4 discusses the proposed framework based on our experiences from these four cases. Finally, Section 5 concludes the paper.

2. Background

The MARTE profile was defined to provide a number of concepts that modelers can use to express relevant properties of RTES, for example related to performance and schedulability. MARTE is meant to replace the previously defined UML profile for Schedulability, Performance, and Time specification (SPT) [13].

At the highest level, MARTE contains three packages. The core package is MARTE Foundations that contains the sub-packages for modeling non-functional properties (NFP package), time properties (Time package), generic resource modeling of an execution platform for RTES (GRM package), and resource allocation (Alloc package). The MARTE Foundations package contains the core elements that are reused by the other two packages of the profile: MARTE design model and RealTime&Embedded Analysing (RTEA). The MARTE design model package contains various sub-packages required for modeling the design of RTES. This includes the packages to support modeling of component-based RTES with the Generic Component Model package (GCM), high-level features for RTES with the High-Level Application Modeling package (HLAM), and for detailed modeling of software and hardware resources with the Detailed Resource Modeling package (DRM). The RTEA package contains further concepts related primarily to modeling for analysis. This includes the Generic Quantitative Analysis Modeling package (GQAM) which provides generic concepts for resource modeling. These concepts are further specialized by the Schedulability Analysis Modeling (SAM) package for modeling properties useful for Schedulability and the Performance Analysis Modeling package (PAM) for modeling properties useful for performance analysis.

3. Industrial Applications of UML/MARTE

This section discusses three UML/MARTE applications in different industrial contexts. For each of the three applications, we provide the case study description, the problem description, the modeling solution, the modeling tool, and the key results of the application. This information will subsequently be used to propose a framework meant to provide guidance to future users of UML/MARTE.

3.1. Architectural Modeling and Configuration with UML/MARTE

3.1.1. Case Study Description

Integrated Control Systems (ICSs) are heterogeneous systems-of-systems, where software and hardware components are integrated to control and monitor physical devices and processes, such as process plants or oil and gas production platforms. FMC Technologies, Inc is a leading global provider of technology solutions for the energy industry. FMC's Subsea Production Systems (SPSs) are large-scale, highly-hierarchical, and highly-configurable ICSs for managing exploitation of oil and gas production fields. One of its key technologies is subsea production systems, used to develop new energy reserves and for managing and improving producing fields. They are composed of hundreds of mechanical, hydraulic, and electrical components and configured software to support various field layouts ranging from single satellite wells to large multiple-well sites (more than 50 wells). The main components of the system are subsea control modules, which contain software, electronics, instrumentation, and hydraulics for safe and efficient operation of subsea tree valves, chokes, and downhole valves.

3.1.2. Problem Description

The research question of this project is to devise a product line architecture modeling methodology, including modeling notations, guidelines and tool support, for the purpose of facilitating the systematic and automated product configuration of ICSs such as FMC's SPSs. The ultimate goal is to improve the overall quality and productivity of the product development lifecycle of ICSs. Specifically, selected/tailored modeling notations of such a methodology should have the following characteristics: (i) It should contain both hardware and software modeling notations and the hardware modeling notations should be expressive enough to capture different types of hardware components and elements; (ii) The interactions between software and hardware components should be captured, such as

the deployment of a software component to its hardware computing resources; (iii) The consistency between hardware and software components should be maintained in the context of supporting configuration; (iv) The variability modeling notation should enable automated configuration and configuration reuse. We have proposed such a produce line architecture modeling methodology, named SimPL [14], to facilitate automated configuration of families of ICSs.

3.1.3. Modeling Solution

In addition to satisfy the modeling requirements described above, there are a number of practical requirements that affect the selection of existing modeling languages: 1) the modeling notation should be easy to learn and apply for industrial partners; 2) the modeling notation should have available tool support. Therefore our modeling solution is based on UML/MARTE, with a minimum extension through the UML profiling extension mechanism.

To facilitate automated configuration, the modeling notation we proposed for modeling the product line architecture uses UML classes, properties, and relationships (i.e., generalization relationships, and several types of association relationships) resulting in base models of hardware and software. In the SimPL methodology we use the following four stereotypes from MARTE to create hardware models and to model software to hardware bindings/allocations. To distinguish between hardware and software classes, any class in the hardware sub-view should be stereotyped by one of the following four MARTE stereotypes: 1) «HwComputingResource» is used to distinguish those electrical hardware components on which software is deployed; 2) «HwDevice» is used to distinguish those hardware devices that are controlled by, or in general interact with, software; 3) «HwComponent» characterizes hardware classes representing hardware components that physically contain other devices and execution platforms; 4) «Assign» models the deployment, allocation, or binding of a structure (e.g., software class) in the software sub-view to a resource (e.g., a hardware component) in the hardware sub-view. UML templates and packages, along with six stereotypes from our newly proposed profile, named SimPL, are used to model the product line architecture.

3.1.4. Modeling Tool

IBM Rational Software Architect (RSA) [15] was used to model the architecture.

3.1.5. Key results

The resulting product-line model contained a total of five views and sub-views and is visualized using 17 class diagrams. The model contains a total of 71 classes, including 46 software classes, 24 classes belonging to the hardware sub-view, and a class representing the topmost element, FMCSytem.

The software sub-view contains configurable software classes related to the selected components of the FMC family, their attributes, their relationships, and supporting containment and taxonomic hierarchies. The hardware sub-view captures a subset of devices (i.e., only those devices that are controlled by software classes captured in the software sub-view), their attributes, and the supporting containment and taxonomic hierarchies. The result is a hardware sub-view with 24 hardware components and devices, including 11 computing resources. Two types of relationships between the software and hardware classes (i.e., allocation of software to hardware and software controlling hardware) are captured in the allocation view.

The variability view contains 22 configuration units, corresponding to 22 configurable classes in software and hardware sub-views. A total of 109 variability points are organized using these configuration units. In addition, a total of 34 dependencies stereotyped with the SimPL profile were created to complete the variability model. A total of 16 OCL constraints are captured in the variability view modeling the dependencies between variability points, mainly the dependencies between variability points introduced by software and those introduced by hardware.

3.2. Model-based Robustness Testing with UML/MARTE

We applied UML/MARTE to support automated, model-based robustness testing of a core subsystem of a video conferencing system developed by Cisco Systems, Norway.

3.2.1. Case Study Description

Our case study is a commercial Video Conferencing System (VCS) called Saturn developed by Cisco Systems Inc, Norway. The core functionality of Saturn manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels and there is also a possibility of transmitting presentations in parallel with audio and video. Presentations can be sent by only one conference participant at a time and all others receive it. In total, Saturn consists of 20 subsystems such as audio and

video subsystems. Each subsystem can run in parallel to the subsystem implementing the core functionality dealing with establishing videoconferences.

3.2.2. Problem Description

Our case study is part of a project aiming at supporting automated, model-based robustness testing of Saturn. A system should be robust enough to handle the possible abnormal situations that can occur in its operating environment and invalid inputs. For example, Saturn should be robust against hostile environment conditions (regarding the network and other communicating VCSs), such as high percentage of packet loss and high percentage of corrupt packets. Saturn should not crash, halt, or restart in the presence of, for instance, a high percentage of packet loss. Furthermore, Saturn should continue to work in a degraded mode, such as continuing the videoconference with lower audio and video quality. In the worst case, Saturn should return to the most recent safe state instead of bluntly stopping execution. Such behavior is very important for a commercial VCS and must be tested systematically and automatically to be scalable.

3.2.3. Modeling Solution

Following, we discuss our modeling solution to support automated robustness testing.

To model the functional behavior, for each subsystem, we modeled a class diagram to capture APIs and state variables. In addition, we modeled one or more state machines to capture the behavior of each subsystem. Due to confidentiality restrictions, we do not provide details of the subsystems. However, on average each subsystem has five states and 11 transitions, with the biggest subsystem having 22 states and 63 transitions. It is important to note that, though the complexity of an individual subsystem may not look high in terms of number of states and transitions, all subsystems run in parallel to each other and therefore the spaces of system states and possible execution interleavings are very large. Saturn's implementation consists of more than three million lines of C code.

Robustness behavior is typically crosscutting many parts of the system functional model and, as a result, modeling such behavior directly within the functional models is not practical since it leads to many redundancies and hence results in large, cluttered models. To cope with this issue, we decided to adopt Aspect-Oriented Modeling (AOM) [16] and more specifically a UML profile for AOM called AspectSM [17]. With it, we model each aspect as a UML state machine with stereotypes (aspect state machine). The modeling of

aspect state machines is systematically derived from a fault taxonomy [17] categorizing different types of faults (faults in the environment such as communication medium and media streams that lead to faulty situations in the environment). Each aspect state machine has a corresponding aspect class diagram modeling different properties of the environment using the MARTE profile, whose violations lead to faulty situations in the environment. More specifically, we used the NFPs package to model properties of the operating environment of Saturn.

Table 1. Summary of features of MARTE and other profiles applied

Robustness Behavior	Stereotypes			Existing MARTE NFPs	Newly introduced NFPs
	NFP	GRM	RobustProfile		
Media Quality	2	1	19	19	2
Network Communication	4	1	13	21	3
Illegal Inputs	-	-	1	2	-

Saturn’s non-functional behaviors consist of five aspect class diagrams and five aspect state machines modeling various robustness behaviors. The largest aspect state machine specifying robustness behavior has three states and ten transitions, which would translate into 1604 transitions in standard UML state machines if AspectSM was not used.

3.2.4. Modeling Tool

IBM RSA was used for modeling class diagrams, UML state machines, and aspect state machines. In addition, we also defined AspectSM in the same tool.

3.2.5. Key Results

Table 1 summarizes the features of the MARTE profile and other profiles, which we used in conjunction with MARTE in our case study. The first column shows various robustness behaviors we modeled in this case study. The first one is related to modeling faulty situations in media, i.e., audio and video, the second behavior is about constraining parameters of events on transitions, which is used to generate test cases exercising the system robustness with illegal inputs, and the third robustness behavior models the behavior of a system in the presence of various network faults. Columns two and three show that we used stereotypes from MARTE NFP and GRM packages. For instance, to model network communication we used four stereotypes from the NFP package (e.g., *NfpType*), whereas we used one stereotype from the GRM package, *CommunicationMedia*.

The fourth column shows the stereotypes from other profiles used in conjunction with MARTE. In our case study, we used stereotypes from RobustProfile [17], which allows modeling various properties of faults (e.g., severity) to assist in defining robustness test strategies. For example, for modeling media quality we used in total 19 stereotypes such as *AudioFault* and *VideoFault* from RobustProfile. The fifth column shows the number of existing NFPs we used that are already defined in MARTE for each of the robustness behaviors. For media quality, we used 19 existing NFPs, e.g., *NPF_Percentage*. The last column shows the number of new NFPs we defined in our case study. For instance, in case of media quality, we defined two new NFPs based on the existing NFPs defined in MARTE, e.g., *PacketLoss* in the case of modeling network communication.

3.3. Testing RTES using UML/MARTE environment models

We applied our approach for model-based testing of RTES to two industrial case studies, involving WesternGeco AS and Tomra AS, both in Norway.

3.3.1. Case Study Description

The case study at WesternGeco is of a very large and complex control system for marine seismic acquisition. The system controls tens of thousands of sensors and actuators in its environment. The timing deadlines on the environment are in the order of hundreds of milliseconds. WesternGeco is a market leader in the field of such seismic systems. The system was developed using Java.

The other case study is an automated bottle-recycling machine developed by Tomra AS. The system under test (SUT) was an embedded device ‘Sorter’, which was responsible to sort the bottles into their appropriate destinations. The system communicated with a number of components to guide recycled items through the recycling machine to their appropriate destinations. It is possible to cascade multiple sorters with one another, which results in a complex recycling machine. The SUT was developed using C.

Both the RTES were running in environments that enforce time deadlines in the order of hundreds of milliseconds with acceptable jitters of a few milliseconds in response time.

3.3.2. Problem Description

RTES typically work in environments comprising large numbers of interacting components. The interactions with the environment can be bound by time constraints. Violating such time constraints, or violating them too often for soft real-time systems, can

lead to serious failures leading to threats to human life or the environment. There is usually a great number and variety of stimuli from the RTES environment with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. Testing all possible sequences of stimuli is not feasible. Hence, systematic automated testing strategies that have high fault revealing power are essential for effective testing of industry scale RTES. The system testing of a RTES requires interactions with the actual environment. Since, the cost of testing in real conditions tends to be high, environment simulators are typically used for this purpose. For the industrial systems of WesternGeco and Tomra, we applied one such approach for black-box system level testing based on the environment models of the systems. These models were used to generate an environment simulator [18], test cases, and obtain test oracles. For test case generation, we applied various testing strategies, including search-based testing and adaptive random testing [19].

3.3.3. Modeling Solution

The environment models were developed using our proposed UML & MARTE Real-time Embedded systems Modeling Profile (REMP) [20]. REMP provided extension to the standard UML class diagram and state machine notations and used the MARTE Time package and GQAM package for modeling timing details and non-deterministic events, respectively. One of the major aims while developing REMP was to keep it as simple as possible. We only used those notations and concepts from UML/MARTE that were essential to model the two industrial case studies. Even though the notation subset was minimal, the goal was to keep REMP generic and applicable to the testing of soft RTES belonging to various domains. This was the motivation to apply the methodology to two case studies that belonged to entirely different domains.

The structural details of a RTES environment were modeled as an environment domain model, which captures the information of various environment components, their properties, and their relationships. For the domain model, we used the UML class diagram notation and annotated class diagram elements with REMP. The behavioral details of the environment were modeled using the state machine notation annotated with REMP. Each environment component has one associated state machine. Such state machines contain information of the nominal behavior of the components, their robustness behavior (e.g., break down of a sensor), and “error states” that should never be reached (e.g., hazardous

situations). If any of these error states is reached, then it implies a faulty RTES. Error states act as the oracle of the test cases, i.e., a test case is successful in triggering a fault in the RTES if an error state of the environment is reached during testing.

3.3.4. Modeling Tool

For initial interactive sessions with experts, we used a sketching tool to model the domain. Later on when we had sufficient details of the system, we used Enterprise Architect for modeling Tomra’s case study (because that was the tool they already used) and IBM RSA for modeling WesternGeco. Later on due to various limitations of Enterprise Architect, we migrated the models to IBM RSA.

3.3.5. Key Results

For Tomra’s case, we had a total of 55 environment components, out of which 43 have a corresponding state machine. For testing, we only focused on a subset of the SUT, for which we only use four of the environment components with a total of 23 states and 38 transitions. For the subset of environment models for WesternGeco’s case, a total of three environment components have a state machine. In total for these components, we modeled 27 states and 46 transitions. In both cases, environment components have a large number of instances during test case execution. For example, in WesternGeco, one of the components could have up to thousands of instances.

From MARTE, we mostly used the concepts of *TimedEvent* and *TimedProcessing* from the Time Package. The MARTE *TimedEvent* concept is used to model timeout transitions, so that it is possible for the time events to explicitly specify a clock (if needed). Each environment component may have its own clock or multiple components may share the same clock for absolute timing. Clocks are modeled using the MARTE’s concept of clocks. According to REMP, if no clock is specified, then by default the notion of time is considered to be according to the physical time. Specifying a threshold time for an action execution or for a component to remain in a state is done using the MARTE *TimedProcessing* concept. This is also a useful concept and can be used, for example, to model the behavior of an environment component when the RTES expects a response from it within a time threshold.

From the GQAM package of MARTE, we used the concept of *GaStep* to model non-determinism. Whenever a timeout transition is labeled with *«gaStep»* and a non-zero value

for the *prob* property, this is interpreted as the probability of taking the transition over the time of the test case execution. This stereotype was used to model scenarios where the modeler wants to specify exact probabilities of an event occurrence. For non-determinism, REMP provides other stereotypes too that give more control to the testing engine to specify the probability of event occurrences.

In our methodology, we chose Java as the action language for writing actions. The decision to choose Java as the action language at the model level is due to the lack of tool support for the UML action language (ALF) [21] at the time our tool was developed. Testers of the SUT are also expected to be more familiar with Java (consistent with our experience of applying the approach in two industrial contexts), rather than with a newly approved, standard language. Moreover, ALF does not provide support for specifying time related actions (e.g., corresponding to the MARTE's concept of an *RTAction* to specify an atomic action). It was also not possible to specify time delays with ALF. Both these concepts were used repeatedly while modeling the environment of both industrial cases.

4. Framework for Applying UML/MARTE in Industry

In this section, we present a framework we devised by combining our experiences in applying UML/MARTE on the industrial problems described above. This framework can help practitioners in future application of UML/MARTE in industrial contexts. At a high level, the framework is presented as a UML activity diagram shown in Figure 1. Following, we briefly discuss each of these activities.

4.1. Perform Domain Analysis (A1)

Each of our industrial applications started from performing a domain analysis. Domain analysis is defined as “the process by which information used in developing software systems within the domain is identified, captured, and organized with the purpose of making it reusable (to create assets) when building new products” [22]. Typically, the domain analysis results in a domain model [23] that captures domain concepts and the relationships among them. A domain model can be described using different notations, UML being a frequently used one. For all the three applications, we used the UML class diagram notation for domain modeling.

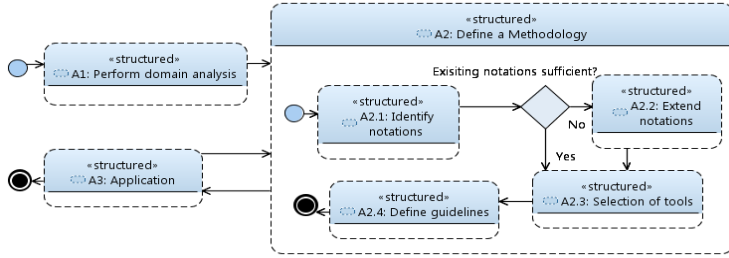


Figure 1. Framework for UML/MARTE applied to industrial applications

The objective of the domain analysis that we performed was different from what is typically presented in OOAD methods [24]. More specifically, our domain analysis is not the start of the software analysis phase but its usage depends on the problem at hand. For architectural modeling, the domain model was later used as a basis to derive the product line architecture modeling methodology, including a UML profile and modeling guidelines. For both the model-based testing projects, the domain analyses resulted in the definition of either environment or system static structure models (as class diagrams), which were used, later on together with state machines, to facilitate automated test-case generation.

To derive the domain models, we followed an iterative process during which we had multiple sessions with our industry partners. In some cases, we initially used sketching tools and simple drawings on white boards for ease of understanding. We started by just capturing the concepts first and later introduced associations and attributes in the domain models. Last, we also added OCL constraints on the domain model concepts. Detailed discussions on how the domain analysis was performed in each of the applications are provided in [6] for architectural modeling, in [17] for robustness testing, and in [20] for model-based testing of RTES.

In all the three cases, the domain analysis was useful in the following ways: (i) It helped us in understanding and specifying (as a domain model) the complexities of large-scale systems having characteristics of multiple disciplines (e.g., electrical, mechanical, and software) and involving multiple stakeholders; (ii) It was instrumental in understanding the needs of industry partners and served as a communication medium with them; (iii) It formed a basis for other activities that we carried out at later stages of the projects, such as defining the modeling methodology and identifying the language and notations for the modeling solution.

4.2. Define a Modeling Methodology (A2)

After performing the domain analysis, we defined a specific modeling methodology to tackle each problem, keeping in mind the requirements of the domain. To apply UML/MARTE in practice, just identifying a set of notations is not sufficient. We need to define a proper process and guidelines, select proper modeling tools, and train the industry partners regarding all these aspects. Following, we discuss the various sub-activities of defining a methodology.

4.2.1. Identify Notations (A2.1)

The first activity for each of the applications was to identify the modeling notations. In all our industrial applications, we carefully selected a subset of UML and MARTE for modeling. The reasons for using UML are as follows: (i) it is a modeling standard; (ii) it has industrial strength tool support ranging from open source (e.g., Papyrus) to commercial (e.g., IBM RSA); (iii) it has sufficient training material available to help train industry partners; (iv) it provides a rich set of notations to model a system from different perspectives; (v) it is extensible for various application domains. Though MARTE is a relatively new profile, we have observed significant progress in tool support and training material available over the last couple of years. Plus it has a rich set of concepts, which can be selected and used for various modeling purposes in the context of real-time, embedded, and concurrent systems.

Despite the above-mentioned advantages, UML is still a challenge to apply in industrial settings without clear objectives and a well-defined methodology. UML is a general purpose, standard modeling language that is meant to cater for different application domain and problems, and is as a result quite large. The entire language is not meant to be used to solve a particular problem in a particular domain. Therefore one of the key requirements to make UML successful in industry is to select a proper subset of the language matching the needs. In our projects, we systematically aimed to identify such a minimal subset. Figure 2 shows the packages of UML that were used for our applications. We used UML class diagrams for modeling the domains for all the industrial case studies. Other notations were selected based on individual needs of the target industrial problem and domain. For architectural modeling we used UML package and class diagram, and for both model-based testing applications, we used UML state machines to model system behavior. In total,

we only used four out of fourteen UML diagrams (including the UML profile diagram that we used to create profiles as part of activity A2.2).

MARTE is a comprehensive UML profile covering different aspects for modeling RTES (Section 2). Similar to UML, the set of concepts provided by MARTE are fairly large to cater to a wide variety of analysis needs and it is also important to clearly identify the required subset of MARTE for a specific problem and domain. Figure 3 shows the six MARTE packages we used (highlighted in grey), a selected subset of the concepts which were used to model our four industrial case studies. In our experience, using UML/MARTE showed to be an adequate combination considering our industrial application domains.

4.2.2. Extend Notations (A2.2)

After we identified the subset of UML and MARTE, the next step was to find out whether the identified notations were sufficient to address our problems. Various steps that we performed in this activity are summarized as an activity diagram in Figure 4. First we evaluated whether the identified MARTE subset was sufficient. If this was not the case, we tried to extend MARTE using the defined constructs (e.g., by adding a new NFP). When required, we further defined guidelines on how to extend MARTE (for example, see [17]) in the future. We then evaluated whether the identified subsets of UML, MARTE, and its extensions were sufficient for our modeling purposes. If this was not the case, we extended UML by creating UML profiles. One of the important decisions was to decide whether to go for a profile or a domain specific language (DSL). In all our cases, we decided to opt for UML profiles over DSL since, in our applications as in many others, minimizing extensions to UML is expected to ease practical adoption and technology transfer. In [25], two main approaches for profile creation are discussed. The first approach directly implements a profile by defining key concepts of a target domain, such as what was done to define SysML [26]. The second approach first creates a conceptual model outlining the key concepts of a target domain followed by creating a profile for the identified concepts, such as what was done to define SPT [13] and MARTE. We used the second approach to define profiles in our context, since it is more systematic as it clearly separates the profile creation process into two distinct stages.

We found the MARTE NFP package and the extension mechanism sufficient for our industrial application of model-based robustness testing. The NFP package provides

different data types such as *NFP_Percentage* and *NFP_DataTxRate*, which are helpful to model properties of the environment, for instance jitter and packet loss in networks. When the built-in data types of MARTE are not sufficient, the open modeling framework of MARTE can be used to define new NFP types by either extending the existing NFPs or by defining completely new NFPs. For instance, we extended MARTE's NFPs and define several properties of the environment when modeling echo in audio streams and modeling synchronization mismatch between audio and video streams coming to a video conferencing system. From our experience in using MARTE, in addition to the advantages of using a standard, we can conclude that the MARTE profile and its open modeling framework were sufficient to model relevant properties of the Saturn operating environment (Section 3.2). However, for our specific problem of robustness testing, we defined a UML profile called RobustProfile [17] to model faults and their properties. In addition, the profile supports the modeling of recovery mechanisms when a fault has occurred and the modeling of states that a system can transition to when it has recovered. Since these features were not part of MARTE, a profile was required.

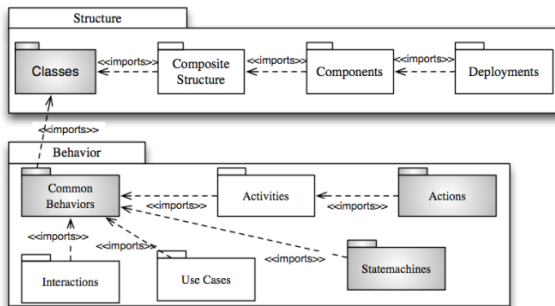


Figure 2. UML packages used in industrial case studies (highlighted in grey)

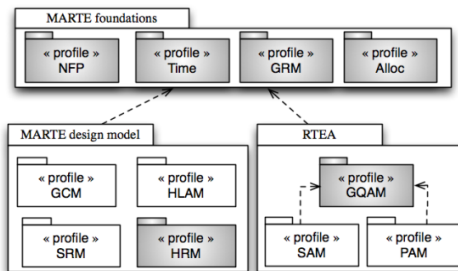


Figure 3. MARTE packages used in industrial case studies (highlighted in grey)

For architecture modeling, we proposed the use of 6 new concepts as stereotypes to extend UML. For model-based robustness testing, we proposed 30 new stereotypes to extend UML and MARTE, and for the environment model-based testing profile, we proposed 8 new stereotypes to extend UML concepts. Overall, we can see that a limited number of stereotypes were required to extend UML for all the three projects. For robustness testing, most of the new stereotypes were based on a fault model and were extending MARTE NFPs.

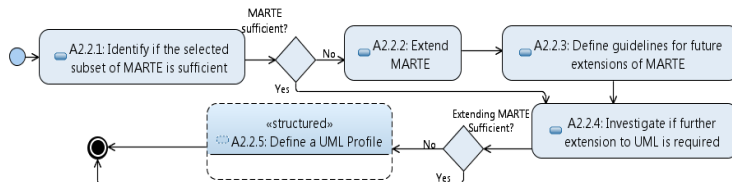


Figure 4. Sub-activities under the activity A2.2 Extend Notations

4.2.3. Tool Selection (A2.3)

An important consideration for the practical adoption of our proposed methodologies in industrial settings is the selection of an adequate modeling tool. This is important since the models developed are meant to support automation (e.g., software test automation). The modeling tool should provide support to export the models in a standard format which can later be processed by other MDE tools (e.g., for model transformations and OCL parsers). According to the MARTE official website [27], the MARTE profile is available in four tools: IBM RSA [15], IBM Rhapsody [28], Papyrus UML [29], and MagicDraw [30].

Among these tools, only IBM RSA and Papyrus UML are EMF-based and hence can be used with other EMF based tools (e.g., Kermeta for model transformations). For Papyrus UML, we faced serious usability problems in modeling state machines, since most of the interface of the tool is based on the assumption that the modeler is aware of the underlying UML metamodel. IBM RSA comes with a high price tag to be used in small to medium sized companies. Even IBM RSA has usability issues, for example, it is not possible to directly link action on a modeling element, such as sending of a signal, in the action code written as part effects in the state machines. Similarly, the MARTE profile is only compatible with RSA version 7.0 and if used with later versions, it does not support the Value Specification Language (VSL) editor. Due to this reason and that a complete parser of VSL was not available at the time we worked on the industrial projects, we used OCL to specify values for NFPs and other MARTE types.

For one of the projects, we also worked with Enterprise Architect [31]. Initially for the domain model we used a sketching tool. It was easier for the industry partners to work with it, because it did not enforce any constraint on the modeling and was good for initial domain modeling. Though Enterprise Architect is cheap and affordable for smaller companies, migrating its models to a form compatible to EMF-based tools is not trivial.

Overall, we found IBM RSA as the most viable modeling tool in terms of usability and its interoperability with third party MBE tools (such as model transformation tools).

4.2.4. Define Guidelines (A2.4)

The next step after tool selection was to define modeling guidelines for each of the methodologies. As discussed earlier, only specifying a set of notations is not sufficient and we need a proper methodology to help modelers determine what to model, in which order, and at what level of detail. The guidelines are not generic and are, to some extent, specific to each domain and application. For example, for the environment model-based testing approach, we defined guidelines to help modelers in identifying test-relevant environment concepts and their relationships [20] in the context of embedded systems. According to our experience, such guidelines are crucial for modelers to correctly and effectively apply our modeling notations.

4.3. Application of Methodology

Once the methodology was defined, numerous training sessions took place, which ranged from acquiring basic UML modeling skills to more advanced methodology specific training. Training was conducted in an interactive manner, where the attendees were given exercises based on their own domain and systems. This last point is very important as people more easily understand and adopt technologies that have shown to apply to their environment.

Training must be complemented by workshops where we model the solution to a representative (sub)problem with them, thus reducing the initial learning curve with respect to the modeling tool and notations. Later, when the first modeling activities are undertaken, mentoring is also required, at least in the initial stages, until a certain level of comfort is attained. A natural tendency is for people to revert to previous practices when faced with a seemingly intractable problem.

4.4. Summary and Discussion

For the three industrial projects, we used the UML class, package and state machine diagrams for modeling the different aspects of software systems. From MARTE, we used concepts from the MARTE Time, NFP, GQAM, Alloc, GRM and HRM packages. Over the years, a number of researchers and industry practitioners have raised the issue that UML is too large [32] [33]. Recently, the same has been written about MARTE [7]. In our opinion and based on our practice, UML and MARTE are meant to provide an encompassing set of modeling notations catering diverse needs. To successfully apply these standards to industrial projects, we need a complete methodology that identifies the subset of UML and MARTE to be used to address specific problems in specific contexts and guidelines to help people apply such standards in a systematic and consistent manner.

A complete methodology based on UML/MARTE should be derived for a specific purpose, to address a particular problem in a particular domain. To do so, we found that a thorough domain analysis is an important step, which, as we discussed in Section 4.1, is a necessary basis not just for the analysis but also to make decisions during other activities. Depending on the complexity of the domain under analysis and the nature of the problem, the domain analysis activities and effort required vary significantly from case to case. The next steps are to carefully select a minimal subset of UML and MARTE notations and if needed, extend MARTE, for example by defining new NFPs, and extend UML by defining a profile. Though the selection of a modeling tool might seem to be a trivial process, in our experience, this can have large impact on adoption by the industry partners. If needed, the modeling tool should be customized based on the modeling notations selected, so that concepts of UML and MARTE that are not relevant are also not visible to the end user. Along with the notations, we found it an essential step to provide a set of modeling guidelines for the end user, which will help her to properly use these notations.

Integrating UML and MARTE can be challenging too, especially when it comes to bridging the semantic gap between the two. For example, when «HwComponent» was used on a class in a class diagram to represent a hardware component, the meaning of its association with another class not carrying any stereotype becomes ambiguous. This is because UML is typically used to model software. Without having any stereotype applied, a class by default implies that it is a software class. Then the association between the hardware component class and the software class should be given a specific meaning, like

the deployment of the software to its hardware platform. In our cases, we address such semantic gaps in our modeling guidelines.

In our experience, there is limited action language support for MARTE concepts, such as time delays between actions and the concept of *RTAction* (e.g., required to model atomic actions). Even in the recently released Action Language for Foundational UML (ALF) [21], such concepts are not supported. We used Java as an action language, which provided the concepts of real-time actions that we required.

For model-based robustness testing of RTES, we defined a profile for modeling faults and their properties, recovery mechanisms, and faulty states. These are based on well-defined fault models in the literature and are applicable to RTES in general. These concepts can be a good addition as they align with the goals of the MARTE profile, though this requires further investigation.

5. Conclusion

Applying Model-based Engineering (MBE) notations and methodologies to real-life industrial problems is a challenging task and very few articles in the research literature report on such experiences. For successful MBE application, a comprehensive methodology for modeling should be adopted that is specific to the problem being solved and adequate for the application domain. This paper discusses our experiences of applying Unified Modeling Language (UML) and the UML profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) to solve three distinct industrial problems related to the use of real-time embedded systems (RTES) in four different industry sectors. The industrial problems that we tackled were related to architectural modeling and configuration, model-based robustness testing, and environment model-based testing of RTES. Based on these experiences, we derived a framework to guide practitioners in their application of UML/MARTE in industrial contexts. This will help practitioners bridge the gap between modeling standards and the modeling needs of industrial RTES.

6. References

- [1] OMG, "Unified Modeling Language Superstructure, Version 2.3, <http://www.omg.org/spec/UML/2.3/>," ed, 2010.
- [2] Artemis. (2011). *Artemis Joint Undertaking - The public private partnership for R & D Embedded Systems*. Available: http://artemis-ju.eu/embedded_systems

- [3] OMG, "Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0, <http://www.omg.org/spec/MARTE/1.0/>," ed, 2009.
- [4] S. Demathieu, A. F. Thomas, A. C. Andre, S. Gerard, and F. Terrier, "First Experiments Using the UML Profile for MARTE," in *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, 2008, pp. 50-57.
- [5] L. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, and T. Yue, "Research-Based Innovation: A Tale of Three Projects in Model-Driven Engineering," Simula Research Laboratory, Technical Report (2011-18)2011.
- [6] T. Yue, L. Briand, B. Selic, and Q. Gan., "Experiences with Model-based Product Line Engineering for Developing a Family of Integrated Control Systems: an Industrial Case Study," Simula Research Laboratory, Technical Report(2012-06)2012.
- [7] H. Espinoza, K. Richter, and S. Gérard, "Evaluating MARTE in an Industry-Driven Environment: TIMMO's Challenges for AUTOSAR Timing Modeling," in *Proceedings of Design Automation and Test in Europe (DATE), MARTE*, 2008.
- [8] S. E. Middleton, A. Servin, Z. Zlatev, B. Nasser, J. Papay, and M. Boniface, "Experiences using the UML profile for MARTE to stochastically model post-production interactive applications," in *eChallenges 2010*, 2010, pp. 1-8.
- [9] *FMC Technologies*. Available: <http://www.fmctechnologies.com>
- [10] *Cisco Inc*. Available: <http://www.cisco.com>
- [11] *WesternGeco*. Available: <http://www.slb.com/services/westerngeco.aspx>
- [12] *Tomra AS*. Available: <http://www.tomra.no>
- [13] (2010). *UML Profile for Schedulability, Performance and Time*. Available: http://www.omg.org/technology/documents/profile_catalog.htm
- [14] R. Behjati, T. Yue, L. Briand, and B. Selic, " SimPL: A Product-Line Modeling Methodology for Families of Integrated Control Systems " *Technical Report 2011-14 (ver.2)*, Simula Research Laboratory, 2012.
- [15] *IBM Rational Software Architect*. Available: <http://www.ibm.com/software/awdtools/architect/swarchitect/>
- [16] R. Yedduladoddi, *Aspect Oriented Software Development: An Approach to Composing UML Design Models*: VDM Verlag Dr. Müller, 2009.
- [17] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," Simula Research Laboratory, Technical Report (2010-03)2010.
- [18] M. Z. Iqbal, A. Arcuri, and L. Briand, "Code Generation from UML/MARTE/OCL Environment Models to Support Automated System Testing of Real-Time Embedded Software," Simula Research Laboratory, Technical Report (2011-04) 2011.
- [19] M. Z. Iqbal, A. Arcuri, and L. Briand, "Automated System Testing of Real-Time Embedded Systems Based on Environment Models," Simula Research Laboratory, Technical Report (2011-19) 2011.
- [20] M. Z. Iqbal, A. Arcuri, and L. Briand, "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies," in *Model Driven Engineering Languages and Systems*. Springer Berlin / Heidelberg, 2010, pp. 286-300.
- [21] OMG, "Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF), Version 1.0 - Beta 1, <http://www.omg.org/spec/ALF/>," ed, 2010.
- [22] P. America, S. Thiel, S. Ferber, and M. Mergel, "Title," unpublished|.

- [23] *Conceptual Model (computer science)*: [http://en.wikipedia.org/wiki/Conceptual_model_\(computer_science\)](http://en.wikipedia.org/wiki/Conceptual_model_(computer_science)).
- [24] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*: Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.
- [25] F. Lagarde, H. Espinoza, F. Terrier, C. André, and S. Gérard, "Leveraging Patterns on Domain Models to Improve UML Profile Definition," in *Fundamental Approaches to Software Engineering*, 2008.
- [26] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*: Tim Weilkiens, 2008.
- [27] *MARTE Tools* Available: <http://www.omgmarTE.org/node/31>
- [28] IBM. (2011). *IBM Rational Rhapsody* Available: <http://www.ibm.com/software/awdtools/rhapsody/>
- [29] *Papyrus UML*. Available: <http://www.papyrusuml.org>
- [30] *MagicDraw*. Available: <http://www.magicdraw.com/>
- [31] *Enterprise Architect*. Available: <http://www.sparxsystems.com/>
- [32] M. Grossman, J. E. Aronson, and R. V. McCarthy, "Does UML make the grade? Insights from the software development community," *Information and Software Technology*, vol. 47, pp. 383-397, 2005.
- [33] J. G. Suess, P. Fritzson, and A. Pop:, "The Impreciseness of UML and Implications for ModelicaML," in *In Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, (EOOLT'2008)*, 2008, pp. 17-26.

A Search-based OCL Constraint Solver for Model-based Test Data Generation

Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, Lionel Briand

In: Proceedings of the 11th International Conference on Quality Software (QSIC 2011), pp. 41-50, IEEE, 2011.

Abstract—Model-based testing (MBT) aims at automated, scalable, and systematic testing solutions for complex industrial software systems. To increase chances of adoption in industrial contexts, software systems should be modeled using well-established standards such as the Unified Modeling Language (UML) and Object Constraint Language (OCL). Given that test data generation is one of the major challenges to automate MBT, this is the topic of this paper with a specific focus on test data generation from OCL constraints. Though search-based software testing (SBST) has been applied to test data generation for white-box testing (e.g., branch coverage), its application to the MBT of industrial software systems has been limited. In this paper, we propose a set of search heuristics based on OCL constraints to guide test data generation and automate MBT in industrial applications. These heuristics are used to develop an OCL solver exclusively based on search, in this particular case genetic algorithm and (1+1) EA. Empirical analyses to evaluate the feasibility of our approach are carried out on one industrial system.

1. Introduction

Model-based testing (MBT) has recently received increasing attention in both industry and academia. MBT promises systematic, automated, and thorough testing, which would likely not be possible without models. However, the full automation of MBT, which is a requirement for scaling up to large systems, requires solving many problems, including preparing models for testing (e.g., flattening state machines), defining appropriate test strategies and coverage criteria, and generating test data to execute test cases. Furthermore, in order to increase chances of adoption, using MBT for industrial applications requires using well-established standards, such as the Unified Modeling Language (UML) and its associated language to write constraints: the Object Constraint Language (OCL) [1].

OCL [1] is a standard language that is widely accepted for writing constraints on UML models. OCL is based on first order logic and is a highly expressive language. The

language allows modelers to write constraints at various levels of abstraction and for various types of models. It can be used to write class and state invariants, guards in state machines, constraints in sequence diagrams, and pre and post condition of operations. A basic subset of the language has been defined that can be used with meta-models defined in Meta Object Facility (MOF) [2] (which is a standard defined by Object Management Group (OMG) for defining meta-models). This subset of OCL has been largely used in the definition of UML for constraining various elements of the language. Moreover, the language is also used in writing constraints while defining UML profiles, which is a standard way of extending UML using pre-defined extension mechanisms.

Due to the ability of OCL to specify constraints for various purposes during modeling, for example when defining guard conditions or state invariants in state machines, such constraints play a significant role when testing is driven by models. For example, in state-based testing, if the aim of a test case is to execute a guarded transition (where the guard is written in OCL based on input values of the trigger) to achieve full transition coverage, then it is essential to provide input values to the event that triggers the transition such that the values satisfy the guard. Another example can be to generate valid parameter values based on the pre-condition of an operation.

Test data generation is an important component of MBT automation. For UML models, with constraints in OCL, test data generation is a non-trivial problem. A few approaches in the literature exist that address this issue. But most of them, either target only a small subset of OCL [3, 4], are not scalable, or lack proper tool support [5]. This is a major limitation when it comes to the industrial application of MBT approaches that use OCL to specify constraints on models.

This paper provides a contribution by devising novel heuristics for the application of search-based techniques, such as Genetic Algorithms (GAs) and (1+1) Evolutionary Algorithm (EA), to solving OCL constraints (covering the entire OCL 2.2 semantics [1]) in order to generate test data. A search-based OCL constraint solver is implemented and evaluated on the first reported, industrial case study on this topic.

The rest of the paper is organized as follows: Section 2 discusses the background and Section 3 discusses related work. In Section 4, we present the definition of distance function for various OCL constructs. Section 5 discusses the case studies and analysis of results of the application of the approach, whereas Section 6 discusses the tool support and Section 7 addresses the threats to validity of our empirical study. Finally, Section 8 concludes the paper.

2. Background

Several software engineering problems can be reformulated as a *search problem*, such as test data generation [6]. An exhaustive evaluation of the entire *search space* (i.e., the domain of all possible combinations of problem variables) is usually not feasible. There is a need for techniques that are able to produce “good” solutions in reasonable time by evaluating only a tiny fraction of the search space. Search algorithms can be used to address this type of problem. Several successful results by using search algorithms are reported in the literature for many types of software engineering problems [7-9].

To use a search algorithm, a *fitness function* needs to be defined. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. The fitness function will be used to guide the search algorithms toward *fitter* solutions. Eventually, given enough time, a search algorithm will find an optimal solution.

There are several types of search algorithms. Genetic Algorithms (GAs) are the most well-known [7], and they are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through *crossover* and *mutation* operators. (1+1)Evolutionary Algorithm (EA) is simpler than GAs, in which only a single individual is evolved with mutation. To verify that search algorithms are actually necessary because they address a difficult problem, it is a common practice to use Random Search (RS) as baseline [7].

3. Related Work

There are a number of approaches that deal with the evaluation of OCL constraints. The basic aim of most of these approaches is to verify whether the constraints can be satisfied. Though most of the approaches do not generate test data, they are still related to our work since they require the generation of values for validating the constraints. These approaches can be adapted for generating test data. In Section 3.1, we discuss the OCL-based constraint solving approaches in the literature. In Section 3.2 we discuss the approaches that use search-based heuristics for testing.

3.1. OCL-based Constraint Solvers

A number of approaches use constraint solvers for analyzing OCL constraints for various purposes. These approaches usually translate constraints and models into a formalism (e.g., Alloy [10], temporal logic BOTL [11], FOL [12], Prototype Verification System (PVS) [13], graph constraints [14]), which can then be analyzed by a constraint analyzer (e.g., Alloy constraint analyzer [15], model checker [11], Satisfiability Modulo Theories (SMT) Solver [12], theorem prover [12], [13]). Satisfiability Problem (SAT) solvers have also been used for the animation of OCL operation contracts (e.g., [16], [17]).

Some approaches are reported in the literature that generates test cases based on OCL constraints. Most of these approaches only handle a small subset of OCL and UML models and are based on formal constraint solving techniques, such as SAT solving (e.g., [3]), constraint satisfaction problem (CSP) (e.g., [18], [19]) and partition analysis (e.g., [5], [4]).

The work presented in [19] is one of the most sophisticated approaches in the literature. However, its focus is on verification of correctness properties, but to achieve this, it also generates an instantiation of the model. The major limitation of that approach is that the search space is bounded and, as the bounds are raised, the CSP faces a combinatorial explosion increase (as discussed in [19]). The task of determining the optimal bounds for verification is left to the user, which is not simple and requires repeated interaction from the user. Models of industrial applications can have hundreds of attributes and manually finding bounds for individual attributes is often impractical. We present the results of an experiment that we conducted to compare our approach with this approach in Section 5.2.

Most of the above approaches are different from our work, since we want to generate test data based on OCL constraints provided by modelers on UML state and class diagrams. These diagrams may be developed for environment models or system models and the modeler should be allowed to use the complete set of standard OCL 2.2 notations. We want to provide inputs for which the constraints are satisfied, and not just verify them. We also want a tool that can be easily integrated with different state-based testing approaches and manual intervention should not be required for every run.

Existing approaches for OCL constraint solving do not fully fit our needs. Almost all of the existing works only support a small subset of OCL. Most of the approaches are only limited to simple numerical expressions and do not handle collections (used widely to specify expressions that navigate over associations). This is generally due to the high expressiveness of OCL that makes the definitions of constraints easier, but their analysis more difficult. Conversion of OCL to a SAT formula or a CSP instance can easily result in

combinatorial explosion as the complexity of the model and constraints increase (as discussed in [19]). For industrial scale systems, as in our case, this is a major limitation, since the models and constraints are generally quite complex. Most of the discussed approaches either *do not support* the OCL constructs present in the constraints that we have in our industrial case study or are not efficient to solve them (see Section 5.2). Hence, existing techniques based on conversion to lower-level languages seem impractical in the context of large scale, real-world systems.

Instead of using search algorithms, another possible approach to cope with the combinatorial explosion faced in solving OCL constraints could be to use hybrid approaches that combine formal techniques (e.g., constraint solvers) with random testing (e.g. [20]). However, we are aware of no work on this topic for OCL and, even for common white-box testing strategies, performance comparisons of hybrid techniques with search algorithms are rare [21].

3.2. Search-based Heuristics for Model Based Testing

The application of search-based heuristics for MBT has received significant attention recently (e.g., [22], [23]). The idea of these techniques is to apply the heuristics to guide the search for test data that should satisfy different types of coverage criteria on state machines, such as state coverage. Achieving such coverage criteria is far from trivial since guards on transitions can be arbitrarily complex. Finding the right inputs to trigger these transitions is not simple. Heuristics have been defined based on common practices in white-box, search-based testing, such as the use of *branch distance* and *approach level* [24]. Our goal is to tailor this approach to OCL constraint solving for test data generation.

4. Definition of the Fitness Function for OCL

To guide the search for test data that satisfy OCL constraints, it is necessary to define a set of heuristics. A heuristic would tell '*how far*' an input data is from satisfying the constraint. For example, let us say we want to satisfy the constraint $x=0$, and suppose we have two data inputs: $x1:=5$ and $x2:=1000$. Both inputs $x1$ and $x2$ do not satisfy $x=0$, but $x1$ is *heuristically* closer to satisfy $x=0$ than $x2$. A search algorithm would use such a heuristic as a fitness function, to reward input data that are closer to satisfy the target constraint.



Figure 1. Example class diagram

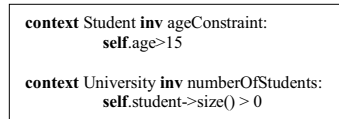


Figure 2. Example constraints

In this paper, to generate test data to solve OCL constraints, we use a fitness function that is adapted from work done for code coverage (e.g., for branch coverage of code written in C [24]). In particular, we use the so called *branch distance* (a function $d()$), as defined in [24]. The function $d()$ returns 0 if the constraint is solved, otherwise a positive value that heuristically estimates how far the constraint was from being evaluated as true. As for any heuristic, there is no guarantee that an optimal solution will be found in reasonable time, but nevertheless many successful results are reported in the literature for various software engineering problems [6].

Notice that, in some cases, we would want the constraints to evaluate to false (e.g., a transition in a state machine that should not be taken). To cope with these cases, we can simply negate the constraint and find data for which the negated constraint evaluates to true.

OCL is a constraint language that is more expressive than programming languages such as C and Java. Therefore, in this paper we extend the basic definition of branch distance to cope with *all* the features of the OCL 2.2 constraint language.

In this section, we give examples of how to calculate the branch distance for various kinds of expressions in OCL, including primitive data types (such as *Real* and *Integer*) and collection-related types (such as *Set* and *Bag*). In OCL, all data types are subtypes of a super type *OCLAny*, which is categorized into two subtypes: primitive types and collection types. Primitive types are *Real*, *Integer*, *String*, and *Boolean*, whereas collection types include *Collection* as super type with subtypes *Set*, *OrderedSet*, *Bag*, and *Sequence*. A constraint can be seen as an expression involving one or more Boolean clauses connected with operators such as *and* and *or*. The truth value of a clause can depend on different types of properties involving variables of different types, such as equalities of integers and comparisons of strings. To explain this, consider the UML class diagram in Figure 1

consisting of two classes: *University* and *Student*. Constraints on the class *University* are shown in Figure 2.

The first constraint states that the age of a *Student* should be greater than 15. Based on the type of attribute *age* of the class *Student*, which is *Integer*, the comparison in the clause is determined to involve integers. The second constraint states that the number of students in the university should be greater than 0. In this case, the *size()* operation is called on collection *student* of the class *Student*, which is defined on collections in OCL and returns an *Integer* denoting the number of elements in a collection. Again, we have a comparison of integers, even though a function such as *size()* is called on a collection.

In the next section, we will discuss branch distance functions based on different types of clauses in OCL.

4.1. Primitive types

A *Boolean* variable b is either true ($d(b)=0$), or false ($d(b)=k$, where for example $k=1$). If the Boolean variable is obtained from a function call, then in general the branch distance would take one of only two possible values (0 or k). However, when such calls belong to the standard OCL operations (e.g., the operation *isEmpty()* called on a collection), then in some cases we can provide more fine grained heuristics (we will specify which ones in more details later in this section).

The operations defined in OCL to concatenate *Boolean* clauses are *or*, *xor*, *and*, *not*, *if then else*, and *implies*. Branch distance for operations on *Boolean* are adopted from [24] and are shown in Table 2. Operations *implies*, *xor*, and *if then else* are syntax sugars that usually do not appear in programming languages, such as C and Java, and can be expressed as combinations of *and* and *or*. The evaluation of $d()$ on a predicate composed by two or more clauses is done recursively, as specified in Table 1.

When a predicate or one of its parts is negated, then the predicate is transformed such as to move the negation inward to the basic clauses, e.g., *not (A and B)* would be transformed into *not A or not B*.

For the data types defined for numerical data such as *Integer* and *Real*, the relational operations defined that return Booleans (and so can be used as clauses) are $<$, $>$, $<=$, $>=$, and $<>$. For these operations, we adopted the branch distance calculation from [24] as shown in Table 2.

In OCL, several other operations are defined on *Real* and *Integer* such as $+$, $-$, $*$, $/$, *abs()*, *div()*, *mod()*, *max()*, and *min()*. Since these operations are used as part of the calculation of

two compared numerical values in a clause, there is no need to define a branch distance for them. For example, considering a and b are of type *Integer* and the constraint $a+b*3<4$, then the operations $+$ and $*$ are used only to define that constraint. The overall result of the expression $a+b*3$ will be an *Integer* and the clause will be considered as a comparison of two values of *Integer* type.

Table 1. Branch distance calculations for OCL's operations for Boolean

Boolean operations	Distance function
Boolean	if true then 0 otherwise k
A and B	d(A)+d(B)
A or B	min (d(A),d(b))
A implies B	d(not A or B)
if A then B else C	d((A and B) or (not A and C))
A xor B	d((A and not B) or (not A and B))

Table 2. Branch distance calculations of OCL's relational operations for numeric data

Relational operations	Distance function
$x=y$	if $\text{abs}(x-y) = 0$ then 0 otherwise $\text{abs}(x-y)+k$
$x \lessdot y$	if $\text{abs}(x-y) \lessdot 0$ then 0 otherwise k
$x < y$	if $x-y < 0$ then 0 otherwise $(x-y)+k$
$x \leq y$	if $x-y \leq 0$ then 0 otherwise $(x-y)+k$
$x > y$	if $(y-x) < 0$ then 0 otherwise $(y-x)+k$
$x \geq y$	if $(y-x) \leq 0$ then 0 otherwise $(y-x)+k$

For the *String* type, OCL defines several operations such as $=$, $+$, $\text{size}()$, $\text{concat}()$, $\text{substring}()$, and $\text{toInteger}()$. There are only three operations that return a Boolean: equality operator $=$, inequality (\lessdot) and $\text{equalsIgnoreCase}()$. In these cases, instead of using k if the comparisons are negative, we can return the value of any string matching distance to evaluate how close two strings are, as for example the edit distance [8].

Enumerations in OCL are treated in the same way as enumerations in programming languages such as Java. Because enumerations are objects with no specific order relation, equality comparisons are treated as basic Boolean expressions, whose branch distance is either 0 or k .

4.2. Collection-Related Types

Collection types defined in OCL are Set, OrderedSet, Bag, and Sequence. Details of these types can be found in [1].

OCL defines several operations on collections. An important point to note is that, if the return type of an operation on a collection is *Real* or *Integer* and that value is used in an expression, then the distance is calculated in the same way as for primitive types as defined in Section 4.4.1. An example is the $\text{size}()$ operation, which returns an *Integer*.

In this section, we discuss branch distance for operations in OCL that are specific to

collections, and that usually are not common in programming languages for expressing constraints/predicates and hence are not discussed in the literature.

4.3. Equality of collections (=)

In OCL constraints, we may need to compare the equality of two collections. To improve the search process by providing a more fine-grained heuristic, we defined a branch distance for comparing collections as shown in Figure 3.

```

if not (A.oclIsKindOf(B))
  d(A=B) := 1
otherwise if A->size() <> B->size()
  d(A=B) := 0.5 + 0.5*n(d (A->size()=B->size()))
otherwise
d(A=B) := 0.5 * sum( n(d(pair)) )/A->size()
where, d(pair) = distance between each paired element in the collection, e.g., d(A.at(i)=B.at(i)) and n is a normalizing function [25], and it is defined as n(x)=x/(x+1). Suppose A and B are two collections in OCL.

```

Figure 3. Branch distance equality of collections

4.4. Operations checking existence of one or more objects in a collection

OCL defines several operations to check existence of one or more elements in a collection such as *includes()* and *excludes()*, which check whether an object exists in a collection or does not exist in a collection, respectively. Whether a collection is empty is checked with *isEmpty()* and *notEmpty()*. Such operations can be further processed for calculation of branch distance to improve the search, as described in Table 3.

Table 3. Branch distance calculation for operations checking objects in collections

Operation	Distance function
includes (object:T): Boolean, where T is any OCL type	$\min_{i=1 \text{ to } \text{self} \rightarrow \text{size}() } d(\text{object} = \text{self.at } i)$
excludes (object:T): Boolean, where T is any OCL type	$\min_{i=1 \text{ to } \text{self} \rightarrow \text{size}() } d(\text{object} <> \text{self.at } i)$
includesAll (c:Collection(T)): Boolean, where T is any OCL type	$\min_{i=1 \text{ to } \text{self} \rightarrow \text{size}() } \min_{j=1 \text{ to } c \rightarrow \text{size}() } d(c.\text{at } i = \text{self.at } j)$
excludesAll(c:Collection(T)): Boolean, where T is any OCL type	$\min_{i=1 \text{ to } \text{self} \rightarrow \text{size}() } \min_{j=1 \text{ to } c \rightarrow \text{size}() } d(c.\text{at } i <> \text{self.at } j)$
isEmpty(): Boolean	$d(\text{self} \rightarrow \text{size}() = 0)$
notEmpty(): Boolean	$d(\text{self} \rightarrow \text{size}() <> 0)$

4.5. Branch distance for iterators

OCL defines several operations to iterate over collections. Below, we will discuss branch distance for these iterators.

The *forAll* iterator operation is applied to an OCL collection and takes as input a *Boolean* expression and determines whether the expression holds for all elements in the collection. For branch distance, we calculate the distance of the *Boolean* expression in

forall. Boolean expression on all elements in the collection is conjuncted. To avoid a bias toward reducing the size of the collection on which the predicate is evaluated, we scale the resulting distance by the number of elements in the collection. The general branch distance function for *forall* is shown in Table 4. For the sake of clarity in the paper, we assume that function $exp(v_1, v_2, \dots, v_m)$ evaluates an expression exp on a set of objects v_1, v_2, \dots, v_m in Table 4. *Self* in the table refers to the collection on which an operation is applied, $at(i)$ is a standard OCL operation that returns the i^{th} element of a collection, and $size()$ is another OCL operation that returns the number of elements in a collection.

Table 4. Branch distance for forall and exists

Operation	Distance function
$forall(v_1, v_2, \dots, v_m exp)$	if $self \rightarrow size() = 0$ then 0 otherwise $\frac{self \rightarrow size() \prod_{i_1=1}^{self \rightarrow size()} \prod_{i_2=1}^{self \rightarrow size()} \dots \prod_{i_m=1}^{self \rightarrow size()} d(exp(self.at(i_1), \dots, self.at(i_m)))}{self \rightarrow size()^m}$
$exists(v_1, v_2, \dots, v_m exp)$	$\min_{i_1, i_2, \dots, i_m \in 1 \text{ to } self \rightarrow size()} d(exp(self.at(i_1), \dots, self.at(i_m)))$
$isUnique(v_1 exp)$	$\prod_{i=1}^{(self \rightarrow size())-1} \prod_{j=i+1}^{(self \rightarrow size())} d(exp(self.at(i)) \neq exp(self.at(j)))$
$one(v_1 exp)$	$d(self \rightarrow select(exp) \rightarrow size() = 1)$

The *exists* iterator operation determines whether a *Boolean* expression holds for at least one element of the collection on which this operation is applied. The distance is computed for each element of the collection on which the *Boolean* expression is applied and the results are disjuncted. The general distance form for *exists* is shown in Table 4. In addition, we also provide branch distance for *isUnique()* and *one()* operations in the same table.

Select, reject, collect, and iterator operations select a subset of elements in a collection. The *select* operation selects all elements of a collection for which a *Boolean* expression is true, whereas *reject* selects all elements of a collection for which a *Boolean* expression is false. In contrast, the *collect* iterator may return a subset of elements, which do not belong to the collection on which it is applied. Since all these iterators return a collection and not a Boolean value, we do not need to define branch distance for them, as discussed in Section 4.4.1.

5. Case Study: Robustness Testing Of Video Conference System

This case study is part of a project aiming at supporting automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn

[26] developed by *Tandberg AS* (now part of Cisco Systems, Inc). Saturn is modeled as a UML class diagram meant to capture information about APIs and system (state) variables, which are required to generate executable test cases in our application context. The standard behavior of the system is modeled as a UML 2.0 state machine. In addition, we used Aspect-oriented Modeling (AOM) and more specifically the AspectSM profile [27] to model robustness behavior separately as aspect state machines. The robustness behavior is modeled based on different functional and non-functional properties, whose violations lead to erroneous states. Such properties can be related to the system or its environment such as the network and other systems interacting with the system. A weaver later on weaves robustness behavior into the standard behavior and generates a standard UML 2.0 state machine. The woven state machine is provided in [27]. This woven state machine is used for test case generation. In this current, simplified case study, the woven state machine has 11 states and 93 transitions. Out of 93 transitions, 73 transitions model robustness behavior and 47 out of 73 are unique, all of them requiring test data that satisfy the constraints to traverse them. All these 47 transitions have change events or triggers. A change event is fired when a condition is met during the operation of a system. An example of such change event is shown in Figure 4. This change event is fired during a videoconference when the synchronization between audio and video passes the allowed threshold. *SynchronizationMismatch* is a non-functional property defined using the MARTE profile, which measures the synchronization between audio and video in time.

```
context Saturn inv synchronizationConstraint:
self.media.synchronizationMismatch.value > self.media.synchronizationMismatchThreshold.value)
```

Figure 4. A constraint checking synchronization of audio and video in a videoconference

In our case study, we target test data generation for model-based robustness testing of the VCS. Testing is performed at the system level and we specifically targeted robustness faults, for example related to faulty situations in the network and other systems that comprise the environment of the SUT. Test cases are generated from the system state machines using our tool TRUST [26]. To execute test cases, we need appropriate data for the state variables of the system, state variables of the environment (network properties and in certain cases state variables of other VCS), and input parameters that may be used in the following UML state machine elements: (1) guard conditions on transitions, (2) change events as triggers on transitions, and (3) inputs to time events. We have successfully used the TRUST tool to generate test cases using different coverage criteria on UML state machines, such as all transitions, all round trip, modified round trip strategy [26].

5.1. Empirical Evaluation

This section discusses the experiment design, execution, and analysis of evaluation of the proposed OCL test data generator.

5.1.1. Experiment Design

We designed our experiment using the guidelines proposed in [7, 28]. The objective of our experiment is to assess the efficiency of search algorithms such as GAs to generate test data by solving OCL constraints. In our experiments, we compared three search techniques: GA, (1+1) EA, and RS. GA was selected since it is the most commonly used search algorithm in search-based software engineering [7]. (1+1) EA is simpler than GAs, but in the previous work in software testing we found that it can be more effective in some cases (e.g., see [9]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [7].

In this paper, we want to answer the following research questions.

RQ1: Are search-based techniques effective and efficient at solving OCL constraints in the models of industrial systems?

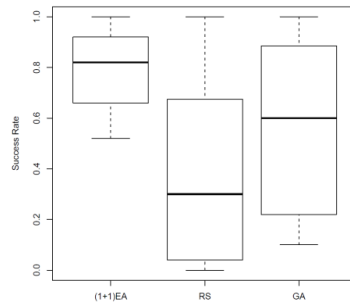


Figure 5. Success rates for various algorithms

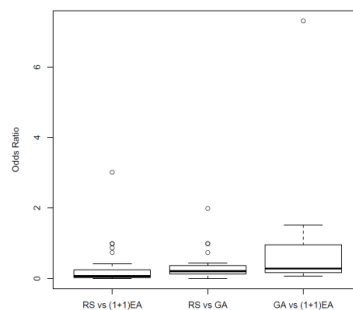


Figure 6. Odds ratio between pairs of algorithms

RQ2: Among the considered search algorithms, which one performs best in solving OCL constraints?

5.1.2. Experiment Execution

We ran experiments for 47 OCL predicates as we discussed in Section 5. The number of clauses in each predicate varies from one to eight and the median value is six. Each algorithm was run 100 times to account for the random variation inherent to randomized algorithms.

A solution is represented as an array of variables, the same that appear in the OCL constraint we want to solve. For GA, we set the population size to 100 and the crossover rate to 0.75, with a 1.5 bias for rank selection. We use a standard one-point crossover, and mutation of a variable is done with the standard probability $1/n$, where n is the number of variables.

We ran each algorithm up to 2000 fitness evaluations on each problem and collected data on whether an algorithm found the solution or not. On our machine (Intel Core Duo CPU 2.20 GHz with 4 GB of RAM, running Microsoft Windows 7 operating system), running 2000 fitness evaluations takes on average 3.8 minutes for all algorithms. Instead of putting a limit to the number of fitness evaluations, a more practical approach would be to run as many iterations as possible, but stopping once a predefined time threshold is reached (e.g., 10 minutes) if the constraint has not been solved yet. The choice of the threshold would be driven by the testing budget. However, though useful in practice, using a time threshold would make it significantly more difficult and less reliable to compare different search algorithms (e.g., accurately monitoring the passing of time, side effects of other processes running at same time, inefficiencies in implementation details).

To compare the algorithms, we calculated their success rate, which is defined as the number of times an algorithm was successful in finding optimal solutions out of the total number of runs.

5.1.3. Results and Analysis

Figure 5 shows a box plot of the success rate of the 47 problems for (1+1) EA, GA, and RS. For each search technique, the box-plot is based on 47 success rates, one for each constraint. The results show that (1+1) EA outperformed both RS and GA, whereas GA outperformed RS. We can observe that, with an upper limit of 2000 iterations, (1+1) EA achieves a median success rate of 80% but GA does not exceed a median roughly 60%. We

can also see that all success rates for (1+1) EA are above 50% and most of them are close to 100%. Constraints with the lowest success rates are seven and eight clauses long. Even taking the lowest success rates for the most difficult constraints (50%), this would entail that with r runs of (1+1) EA, we would achieve a success rate of $1 - (1 - 0.5)^r$. For example, with $r = 7$, we would obtain a success rate above 99%. This entails a computation time of approximately $3.8 * 7 = 27$ minutes. Given that we use a slow prototype (EyeOCL) for OCL expression analysis and that we could parallelize the search, our results suggest that our approach is effective, efficient, and therefore practical, even for difficult constraints (RQ1).

To check the statistical significance of the results, we performed Fisher's exact test between each pair of algorithms based on their success rates for the 47 constraints. Due to space limitations, we do not present p-values for each problem and each pair of algorithms. In summary, we observe that for 105 times out of 141 ($47 * 3$, where 3 represent the number of algorithm pairs), results were significant at the 0.05 level. We also carried out a *paired* Mann-Whitney U-test (paired per constraint) on the distributions of the success rates for the three algorithms. In all the three distribution comparisons, p-values were very close to 0, and hence showing a strong statistical difference among the three algorithms when applied on all the 47 constraints (although on some constraints there is no statistical difference, as the 141 Fisher's exact tests show).

In addition to statistical significance, we also assessed the magnitude of the improvement by calculating the effect size in a standardized way. We used odds ratio [28] for this purpose, as the results of our experiments are dichotomous. Figure 6 shows box plots of odds ratio for pairs of algorithms for the 47 constraints. Between RS and (1+1) EA (the first column in Figure 6), the value of odds ratio is less than one, thus implying that (1+1) EA has more chances of success than RS. The odds ratio between RS and GA is also similar. Therefore, there is strong evidence to claim that (1+1) EA is significantly more successful than the other analyzed algorithms since, in most of the cases, the odds ratios comparing GA and RS with (1+1) EA (first and third column in Figure 6) show values not only lower than one, but also very close to zero (RQ2).

To check the complexity of the problems, we repeat the experiment on the negation of each of the 47 predicates. All algorithms managed to find solutions for all these problems very quickly. Most of the time and for most of the problems, each algorithm managed to find solutions in a single iteration. This result confirmed that the actual problems we targeted with search were not easy to solve.

In practice, given a time budget T , we recommend running (1+1) EA for as many iterations as possible. An alternative is to run the algorithms several times (e.g., r , so each run with budget T/r) but this is expected to be less effective as no information is reused between runs. But, in our experiments, this latter technique is already extremely effective (99% success rate with seven runs in the worst case).

5.2. Comparison with UMLtoCSP

UMLtoCSP [19] is the most widely used and referenced OCL constraint solver in the literature. To check the performance of UMLtoCSP to solve complex constraints such as the ones in our current industrial case study, we conducted an experiment. We selected the 10 most complex constraints (based on the number of clauses in a constraint) from our industrial application, which comprises constraints ranging from six to eight clauses (we did not analyze all the 47 constraints because, as we will show, these experiments took substantial computational time). An example of such constraint, modeling a change event on a transition of Saturn's state machine, is shown in Figure 7. This change event is fired when Saturn is successful in recovering the synchronization between audio and video. Since UMLtoCSP does not support enumerations, we converted each enumeration into an Integer and limited its bound to the number of literals in the enumeration. We also used the MARTE profile to model different non-functional properties, and since UMLtoCSP does not support UML profiles, we explicitly modeled the used subset of MARTE as part of our models. In addition, UMLtoCSP does not allow writing constraints on inherited attributes of a class, so we modified our models and modeled inherited attributes directly in the classes. We set the range of Integer attributes from 1 to 100.

We ran the experiment on the same machine as we used in the experiments reported in the previous section. Though we let UMLtoCSP address each of the selected constraints for *10 hours each*, it was not successful in finding any valid solution. A plausible explanation is that UMLtoCSP is negatively affected by the state explosion problem, a common problem in real-world industrial applications such as the one from Tandberg/Cisco used in this paper. In contrast, even in the worst case, our constraint solver managed to solve each constraint within at most 27 minutes, as we have reported in the previous section.

6. Tool Support

We developed a tool in Java that interacts with an existing library, an OCL evaluator called

EyeOCL [29]. EyeOCL is a Java component that provides APIs to parse and evaluate an OCL expression based on an object model. Our tool implements the calculation of branch distance as discussed in Section 4 for various expressions in OCL. To calculate branch distance for an OCL expression, we send this expression for parsing to EyeOCL and obtain a parse tree of the expression. We manipulate the parse tree and call EyeOCL with the current set of values for variables in the expression and calculate the branch distance. The search algorithms employed in this paper were implemented in Java as well.

7. Threats to Validity

To reduce *construct validity* threats, we chose the measure *success rate*, which is comparable across all three algorithms ((1+1) EA, GA and RS) that we used. Furthermore, we used the same stopping criterion for all algorithms, i.e., number of fitness evaluations. This criterion is comparable across all the algorithms that we studied because each iteration requires updating the object diagram in EyeOCL and evaluating a query on it. This time is same for all the algorithms, and it is rather expensive (approximately, 0.114 second per iteration).

The most probable *conclusion validity* threat in experiments involving randomized algorithms is due to random variation. To address it, we repeated experiments 100 times to reduce the possibility that the results were obtained by chance. Furthermore, we perform Fisher exact test to compare proportions to determine statistical significance of results. We chose Fisher's exact test because it is appropriate for dichotomous data where proportions must be compared, thus matching our case [28]. To determine practical significance of results, we measure the effect size using the odds ratio of success rates across search techniques.

A possible threat to *internal validity* is that we have experimented with only one configuration setting for the GA parameters. However, these settings are in line with common guidelines in the literature and our previous experience on testing problems.

In the empirical comparisons with UMLtoCSP, there is the threat that we might have wrongly configured it. To reduce the probability of such an event, we contacted the authors of UMLtoCSP who were very helpful in ensuring its proper use.

We ran our experiments on an industrial case study to generate test data for 47 different OCL constraints, ranging from simpler constraints having just one clause to complex constraints having eight clauses. Although the empirical analysis is based on a

real industrial system and not on small artificial problems (as most work in the literature [11], [13], and [16]), our results might not generalize to other case studies. However, such threat to *external validity* is common to all empirical studies.

```

context Saturn inv synchronizationConstraint:
  self.systemUnit.NumberOfActiveCalls > 1 and
  self.systemUnit.NumberOfActiveCalls <= self.systemUnit.MaximumNumberOfActiveCalls) and
  self.media.synchronizationMismatch.unit = TimeUnitKind::s and
  (
    self.media.synchronizationMismatch.value >= 0 and
    self.media.synchronizationMismatch.value <=
      self.media.synchronizationMismatchThreshold.value
  )
  and self.conference.PresentationMode = Mode::Off and
  self.conference.call->select(call |
    call.incomingPresentationChannel.Protocol <> VideoProtocol::Off->size() = 2 and
    self.conference.call->select(call |
      call.outgoingPresentatiionChannel.Protocol <> VideoProtocol::Off->size()=2
  )

```

Figure 7. A change event checking which is fired when synchronization between audio and video is within threshold

From our analysis of UMLtoCSP, we cannot generalize our results to traditional constraint solvers in general when applied to solve OCL constraints. However, empirical comparisons with other constraints solvers were not possible because, to the best of our knowledge, UMLtoCSP is not only the most referenced OCL solver but also the only one that is publically available.

8. Conclusion

In this paper, we presented a search-based constraint solver for the Object Constraint Language (OCL). The goal is to achieve a practical, scalable solution to support test data generation for Model-based Testing (MBT). Existing OCL constraint solvers have one or more of the following problems that make them difficult to use in industrial applications: (1) they support only a subset of OCL; (2) they translate OCL into formalisms such as first order logic, temporal logic, or Alloy, and thus are relying on non-standard technologies and result into combinatorial explosion problems. These problems limit their practical adoption in industrial settings.

To overcome the abovementioned problems, we defined a set of heuristics based on OCL constraints to guide search-based algorithms (genetic algorithms, (1+1) EA) and implemented them in our search-based OCL constraint solver. More specifically, we defined branch distance functions for various types of expressions in OCL to guide search algorithms. We demonstrated the effectiveness and efficiency of our search-based

constraint solver to generate test data in the context of the model-based, robustness testing of an industrial case study of a video conferencing system. Even for the most difficult constraints, with research prototypes and no parallel computations, we obtain test data within 27 minutes in the worst case and in less than 4 minutes on average.

As a comparison, we ran the 10 most complex constraints on one well-known, downloadable OCL solver (UMLtoCSP) and the results showed that, even after running it for 10 hours, no solutions could be found. Similar to all existing OCL solvers, because it could not handle all OCL constructs, we had to transform our constraints to satisfy UMLtoCSP requirements.

We also conducted an empirical evaluation in which we compared three search algorithms using two statistical tests: Fisher's exact test between each pair of algorithms to test their differences in success rates for each constraints and a paired Mann-Whitney U-test on the distributions of the success rates (paired per constraint). Results showed that (1+1) EA was significantly better than GA, which itself were significantly better than random search. Notice that in both empirical evaluations, the execution times were obtained on a regular PC.

Future work will consider hybrid approaches, in which traditional constraint solver techniques will be integrated with search algorithms, with the aim to overcome the current limitations that both approaches have and exploit the best of both worlds.

Acknowledgement

The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the ITEA-2 project called VERDE. We thank Marius Christian Liaaen (Tandberg AS, part of Cisco Systems, Inc) for providing us the case study.

9. References

- [1] (2010). *Object Constraint Language Specification, Version 2.2*. Available: <http://www.omg.org/spec/OCL/2.2/>
- [2] N. Holt, B. Anda, K. Asskildt, L. Briand, J. Endresen, and S. Frøystein, "Experiences with Precise State Modeling in an Industrial Safety Critical System," presented at the Critical Systems Development Using Modeling Languages, CSDUML'06, 2006.
- [3] L. v. Aertryck and T. Jensen, "UML-Casting: Test synthesis from UML models using constraint resolution," presented at the Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003), 2003.

- [4] M. Benattou, J. Bruel, and N. Hameurlain, "Generating test data from OCL specification," 2002.
- [5] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test case automate generation from uml sequence diagram and ocl expression," presented at the International Conference on computational Intelligence and Security, 2007.
- [6] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," King's College, Technical Report TR-09-032009.
- [7] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," *IEEE Transactions on Software Engineering*, vol. 99, 2009.
- [8] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 16, pp. 175-203, 2006.
- [9] A. Andrea, "Longer is Better: On the Role of Test Sequence Length in Software Testing," International Conference on Software Testing, Verification, and Validation, 2010.
- [10] B. Bordbar and K. Anastasakis, "UML2Alloy: A tool for lightweight modelling of Discrete Event Systems," presented at the IADIS International Conference in Applied Computing, 2005.
- [11] D. Distefano, J.-P. Katoen, and A. Rensink, "Towards model checking OCL," presented at the ECOOP-Workshop on Defining Precise Semantics for UML, 2000.
- [12] M. Clavel and M. A. G. d. Dios, "Checking unsatisfiability for OCL constraints," presented at the In the proceedings of the 9th OCL 2009 Workshop at the UML/MODELS Conferences, 2009.
- [13] M. Kyas, H. Fecher, F. S. d. Boer, J. Jacob, J. Hooman, M. v. d. Zwaag, T. Arons, and H. Kugler, "Formalizing UML Models and OCL Constraints in PVS," *Electron. Notes Theor. Comput. Sci.*, vol. 115, pp. 39-47, 2005.
- [14] J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. ster, "Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars," *Electron. Notes Theor. Comput. Sci.*, vol. 211, pp. 159-170, 2008.
- [15] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: the alloy constraint analyzer," presented at the Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland, 2000.
- [16] M. Krieger and A. Knapp, "Executing Underspecified OCL Operation Contracts with a SAT Solver," presented at the 8th International Workshop on OCL Concepts and Tools., 2008.
- [17] M. P. Krieger, A. Knapp, and B. Wolff, "Automatic and Efficient Simulation of Operation Contracts," presented at the 9th International Conference on Generative Programming and Component Engineering, 2010.
- [18] B. K. Aichernig and P. A. P. Salas, "Test Case Generation by OCL Mutation and Constraint Solving," presented at the Proceedings of the Fifth International Conference on Quality Software, 2005.
- [19] J. Cabot, R. Claris, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," presented at the Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, 2008.
- [20] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 263-272, 2005.

- [21] K. Lakhotia, P. McMinn, and M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN," *Journal of Systems and Software*, vol. 83, pp. 2379-2391.
- [22] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart, "GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths," *Advanced Design and Manufacture to Gain a Competitive Edge*, pp. 147-156, 2008.
- [23] R. Lefticaru and F. Ipate, "Functional Search-based Testing from State Machines," presented at the Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, 2008.
- [24] P. McMinn, "Search-based software test data generation: a survey: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 14, pp. 105-156, 2004.
- [25] A. Arcuri, "It Does Matter How You Normalise the Branch Distance in Search Based Software Testing," presented at the Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation.
- [26] S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. C. Briand, "Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies," Simula Research Laboratory, Technical Report (2010-01)2010.
- [27] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," Simula Research Laboratory, Technical Report (2010-03)2010.
- [28] A. Arcuri and L. Briand., "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," presented at the International Conference on Software Engineering (ICSE), 2011.
- [29] M. Egea, "EyeOCL Software," ed, 2010.