Model-based Testing for Reactive Systems

Intelligent Approaches

Annamária Szenkovits

Supervisor: Prof. Dr. Horia F. Pop

Summary of the PhD Thesis

Department of Mathematics and Computer Science
Babeş-Bolyai University Cluj-Napoca
Kogalniceanu str. 1, RO-400084

July 2017

Contents

Keywords. model-based testing, automated test generation, evolutionary testing, statistical testing, reactive systems

1 Introduction

Testing is a crucial step in the software development life-cycle. It is common to dedicate at least 50% of the project resources to this step [Beizer, 1990]. Therefore the problem of finding efficient ways to automate different aspects of software testing received a lot of attention in the scientific community [Ammann and Offutt, 2008].

Testing is even more crucial for reactive systems, as they are very often critical (e.g., in embedded systems). Reactive systems present additional challenges in this regard, because of the feedback loop: a reactive system acts on its environment, which in turn acts on the system. Realistic input sequences need to be generated on-the-fly, by executing the system in a simulated environment. Moreover, environments are intrinsically stochastic, because they may vary a lot, and also because they are not perfectly known. Lutin [Raymond et al., 2008] is a language dedicated to the programming of such environment simulators; it allows one to model stochastic reactive systems. Lutin programs perform a guided random exploration of the system under test (SUT) environment state-space. This exploration is parametrised by weights, that define probabilities to various behaviors to occur. Such probabilities are not always easy to define. The central idea of this article is to take advantage of evolutionary algorithms to optimize these parameters automatically.

1.1 Contributions of this thesis

The main contribution of this thesis can be summarized as follows:

- We propose a testing framework optimized for reactive systems. In this framework, a non-deterministic, executable environment model is created based on the system specifications. This environment model is used for stimulating the SUT, but also to automatically generate test inputs. The language proposed for creating the environment model is Lutin [Raymond et al., 2008], an automatic test generator for reactive programs developed by the Verimag research lab, which enables us to perform guided random exploration of the environment's state space.

- Guidance (in test case generation) can be accomplished by making use of certain in-built features of the Lutin language that effectively offer the possibility to parameterize the process which generates test inputs during testing. To achieve high

model coverage and to target specific internal state regions of the SUT we fine-tune the Lutin environment parameters so that the test cases generated and executed against the SUT would cover the structure of the SUT code as much as possible.

- In order to achieve the fine-tuning of the Lutin environment, we proposed an approach involving two evolutionary algorithms: Differential Evolution [Storn and Price, 1997] and an adaptive Differential Evolution [Zhang and Sanderson, 2009]. We used the parameters of the Lutin environment to create the members of the population, and the fitness of an individual was represented by the coverage of the SUT. We used two different coverage metrics: MC/DC and decision coverage.

- To evaluate the effectiveness of the testing framework proposed, we ran experiments on both toy problems and a real-world, industrial system from the domain of railway automation.

2 Model-based testing and reactive systems: Foundations, state of the art and challenges

In this chapter, the main steps of the process of model-based testing are presented, as well as the benefits of this testing technique over other testing methods (e.g. manual testing, script based testing, capture/replay testing). In addition, the most widely used modeling notations and test selection criteria are described. Afterwards, the main characteristics and behavior of reactive systems is summarized, with emphasis on the most important problems that arise when we want to perform testing on such systems. Finally, we give an outline of the main questions addressed in this thesis.

2.1 The process of model-based testing

The main idea of model-based testing is to derive a model based on the abstractions of the system under test (SUT) and/or its environment and then to generate test cases based on this model. The process involves the following main steps:

1. Create a model of the SUT, its environment, or both.

2. Generate abstract tests based on the model.

3. Concretize the abstract tests so they can be executed.

4. Execute the concretized tests on the SUT and assign verdicts to the outcome.

5. Analyze the test results. [Utting and Legeard, 2007]

Thus, the first step is building a model of the SUT, its environment, or both the SUT and the environment, based on the available requirements or specification documents. The second step consists of defining the test selection criteria. A selection criterion can describe a certain functionality of the SUT (in case of test selection criteria that are derived based on the requirements), it can relate to the structure of the model (in case of state and transition based coverage criteria, respectively), or to stochastic characterizations (randomness or user profiles). During the third step, the test selection criteria are formalized and transformed into test case specifications. Next, an automatic test case generator derives a test suite based on the model of the SUT and a test case specification. Finally, the test cases are run. Because the model and the SUT are on different levels of abstractions, the input part of a test case must be concretized first. This step is performed by a component called *adaptor*. At the end, the output of the SUT is compared with the expected output, and the result of this comparison is called *verdict*. The verdict can take the following outcomes: *pass*, *fail* or *inconclusive*.

Given this definition of model-based testing, the process is basically the automation of black-box test design. Thus, when applying model-based testing, the tester has to generate executable test cases that include oracle information, i.e., expected output values of the SUT.

2.1.1 Notations for modeling

In this section, we give an overview of the most frequently used types of modeling notations. In addition, we point out to which group of modeling techniques do SCADE and Lutin belong to.

According to [Utting et al., 2012], the main notations for modeling can be grouped as follows.

- **Pre/post (or state based) notations**: The system is modeled as a collection of variables which represent the internal state of the system at a given time. In addition to the variables, operators that can modify the variables are also described. Instead of using programming language code to define the operators, preconditions and postconditions are used. Examples of pre/post notations include abstract machines in B [Abrial, 1996], the UML Object Constraint Language (OCL) [Warmer and Kleppe, 2003], the Java Modeling Language (JML) [Leavens and Cheon], VDM [Jones, 1990] [Fitzgerald et al., 2005] and Z.

- **Transition based notations**: As suggested by the name, these notations describe the transitions between the different states of the system. Transitions based notations are typically graphical node-and-arch notations, e.g.: finite state machines, state charts (e.g., UML State Machines, Simulink State flow charts), labeled transition systems, and I/O (input/output) automata.

- **Functional notations**: In the case of functional notations, the system is represented as a collection of mathematical functions.

- **Operational notations**: When using operational notations, the system is modeled as a set of executable processes, executing in parallel. These notations are especially suited for describing distributed systems and communication protocols, e.g.: Petri net notations.

- **Statistical notations**: The SUT is represented as a probability model of the events and input values. Although these notations are suitable for modeling events and their input values, they are weak at predicting the expected output of the SUT, i.e., the automatic generation of oracles. For modeling expected usage profiles, one of the most successful methods are Markov chains.

- **Data flow notations**: Rather than modeling the control flow of the system, data flow notations represent the flow of the data through the system. For example, Lustre and the block diagram notations that are used in Matlab Simulink and SCADE for the modeling of continuous systems use data flow notations.

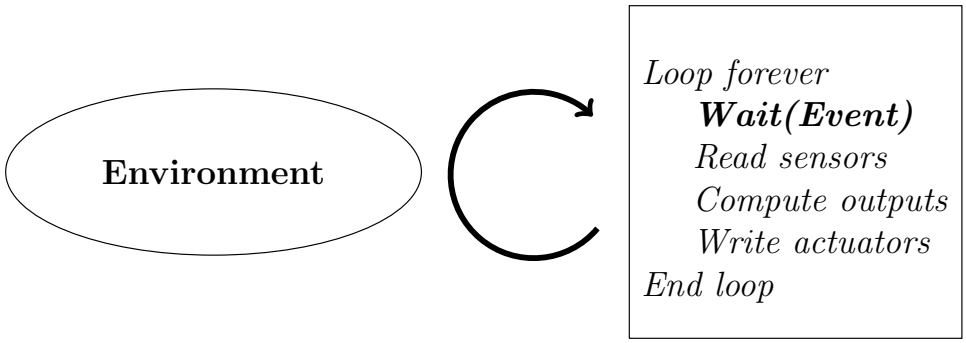

Figure 1: Event-triggered model of a reactive system. A coffee vending machine is a good example for an event-driven reactive system. The customer can execute certain events like selecting drinks or ingredients and inserting money. These events drive the functioning of the machine.

2.2 Main characteristics of reactive systems

Contrary to transformational systems, *reactive system* are systems that have a cyclic behavior, and permanently interact with their environment. Starting from some initial input, they will continue to interact with their environment during the course of their execution. The term *reactive system* was first introduced by David Harel and Amir Pnueli [Harel and Pnueli, 1985].

To describe the behavior of reactive systems, there are two main models available. As illustrated by figures 1 and 2, we can speak about an event-triggered and a time-triggered model.

3 Optimizing the automated test input generation with intelligent methods

In the following, we briefly describe the modeling languages and tools used to create the testing framework. Then, we present the approaches proposed in which we used different search algorithms to optimize test input generation in the framework.

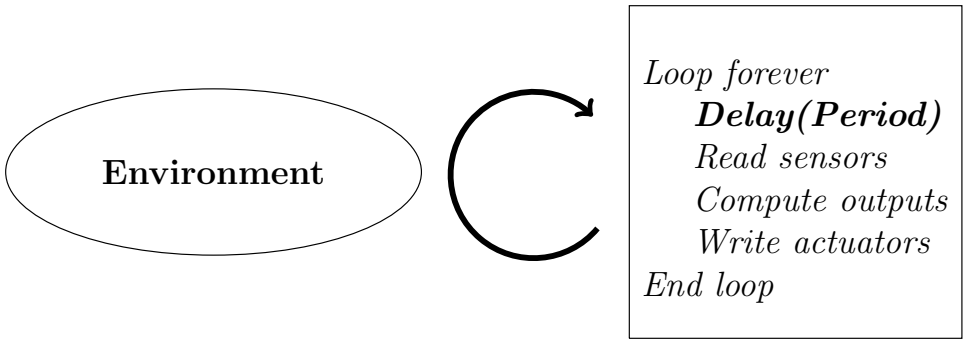

Figure 2: Time-triggered model of a reactive system. For example, a heat controller that has to keep its environment's temperature in a given interval is a time-triggered reactive system. Once in each time unit, it reads the temperature of the environment via its sensors, updates its internal model, then decides whether to increase or decrease the temperature

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

Figure 3: Example of Lustre code demonstrating the usage of operator *pre*.

3.1 Languages and tools used for automated test generation of reactive systems

3.1.1 Lustre: the backbone of SCADE and Lutin

Lustre is a functional language structured on so-called **nodes**. A node represents a program or a subprogram and it operates on **streams**: a finite or infinite sequence of values of a given type. A Lustre program has a cyclic behavior, so that at the nth iteration of the program, all the streams in the program take their nth value. A node generates one or more output parameters based on one or more input parameters. All these parameters are streams. Fig. 3 shows an example of a Lustre node.

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

Figure 4: Lutin node generating an infinite sequence of integers: 42 with a probability of 0.75 and 1 with probability 0.25.

3.1.2 SCADE language and SCADE Suite toolset for modeling safety-critical systems

SCADE is a graphical modeling language, widely-used for safety critical systems, especially for designing avionic and railway systems. SCADE uses essentially two different modeling notations: data flows and state machines (for a more detailed description of the different modeling notations see section 2.1.1). One can define SCADE operators by combining data flows with state machines.

In the following, we give some examples for the data flow and state machine notations.

3.2 Lutin and non-deterministic environment models

Lutin is an automatic test generator for reactive systems that focuses on functional testing [Jahier, 2004].

A Lutin *node* is a data-flow program that transforms a sequence of input tuples (made of Boolean, integer, or real values) into a sequence of output tuples, exactly as SCADE or Simulink block-diagrams do. The main difference with a SCADE or Simulink node is that they are made of a set of equations that have (thanks to some syntactic restrictions) exactly one solution, whereas a Lutin node is made of a set of (linear) constraints that can have any number of solutions.

In order to express different scenarios, Lutin also has control structures based on regular operators: the sequence (`fby`: pronounce followed by), the choice (`|`), and the Kleene star (`loop`). For example, the (input-free) Lutin node in Fig. 4 will generate an infinite sequence of integers with 0.75 probability for 42 and 0.25 probability for 1.

3.3 Environment-model based testing framework with Evolutionary Techniques

Evolutionary algorithms (EA) are powerful optimization tools, they can be easily adapted to specific optimization tasks and have good applicability in different fields such as engineering, robotics, biology, economics, etc. EA is inspired by biological evolution, using mechanisms such as reproduction, mutation, recombination and selection.

In the class of evolutionary algorithms Genetic Algorithms (GAs) are one of the most popular and best known techniques for solving optimization problems [Inza et al., 1999; Sagarna et al., 2003]. GAs are a population based search method and they involve the following main steps:

Algorithm 1 GEA

1: Set of individuals or candidate solutions to the optimization problem is **created**. (This set is referred to as a population.)
2: Promising individuals are **selected** from the population based on a fitness function.
3: A new population is generated based on the selected individuals using **crossover** and **mutation** operators.

3.3.1 Components of the testing framework

To investigate the way evolutionary methods are applicable for reactive systems, we proposed to extend the Lutin automatic test generator tool for reactive systems with an evolutionary testing module. The testing framework proposed consisted of the following components, as illustrated by Fig. 5: the SUT, environment model and MTC analyzer.

3.3.2 Population representation (parameters optimized)

We parameterized the weights of the Lutin choice operators in the environment model and let the DE algorithm optimize these weights. We restricted each weight to the range between 1 and 100. For the choice operators with two branches, we assigned weight p to the first branch and $100 - p$ to the second one (e.g. Fig. 6).

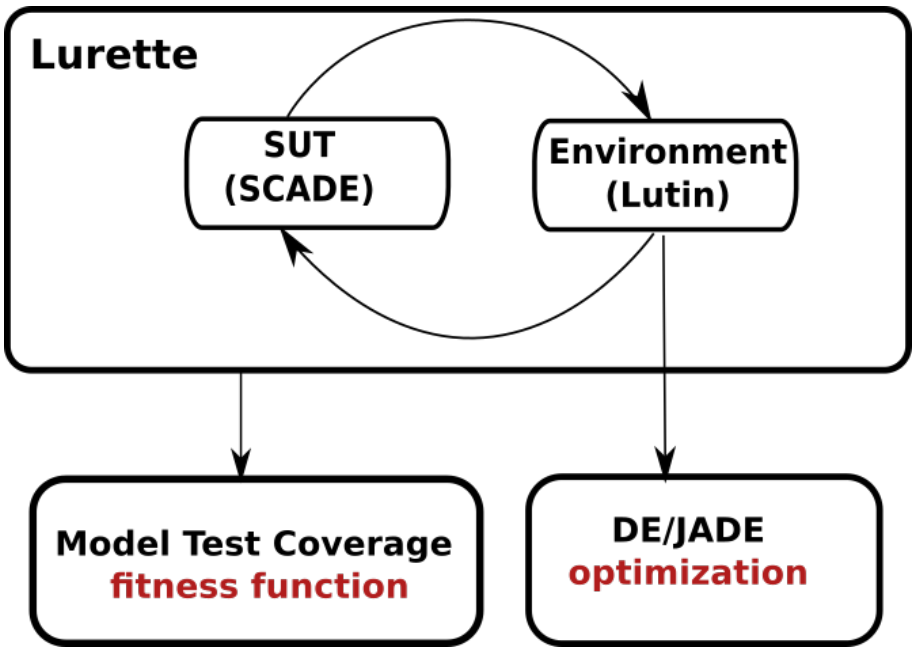

Figure 5: Components of the test framework: The **SUT** as a C code generated from a SCADE model with SCADE Suite KCG, the **environment** model expressed in Lutin and the **MTC analyzer**. The SUT and environment are in continuous interaction with each other and have a cyclic behavior. The execution of the SUT and environment is done by the Lurette tool. DE/JADE-driven optimization is added to the framework to compute some parts of the Lutin environment and improve the automated test input generation.

```
-- pressing and releasing of the button
node button () returns (bac: bool) =
  loop {
    |p: bac = true
    |100-p: bac = false
  }
```

Figure 6: Simulating the pushing of a button. The simulation is realized with Lutin's random choice operator. Weight p assigned to the branches is optimized by DE.

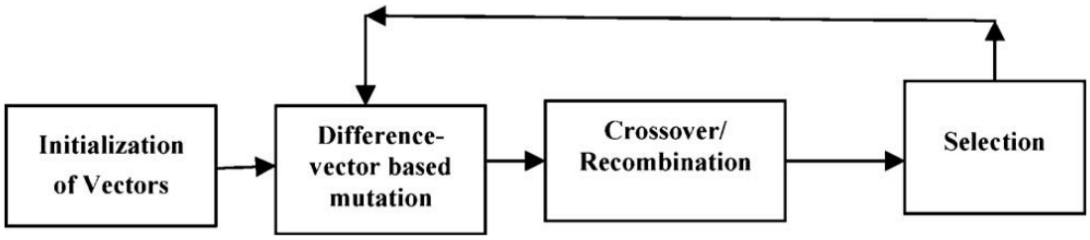

Figure 7: Main steps of the DE algorithm.

3.3.3 Optimizing the environment model with Differential Evolution

Differential Evolution (DE) [Storn and Price, 1997] is a stochastic, population-based optimization algorithm, with good results in many real-world optimization problems, even in non-continuous or dynamic environments (some applications and description can be found in [Das and Suganthan, 2011]). DE is very popular because of its simplicity, speed, and robustness. The main steps of the algorithm are shown in Fig. 7.

Operators. The initial population was selected randomly from the admissible parameter ranges. During each iteration, for each individual l from the population, an offspring $O[l]$ was created using the scheme presented in Algorithm 3, where $U(0, x)$ is a uniformly distributed number between 0 and x, CR denotes the crossover ratio, dim is the number of parameters of the problem, while F is the scaling factor.

The main steps of the algorithm are outlined in Algorithm 2.

Algorithm 2 DE [Mihoc et al., 2016]

Randomly generate initial population P_0 of solutions;
while (not termination condition) **do**
  for each $l = \{1, ..., population\ size\}$ **do**
    create offspring $O[l]$ from parent l;
    if $O[l]$ is better then parent j **then**
      $O[l]$ replaces parent j;
    end if
  end for
end while

Algorithm 3 DE - the DE/*rand/1/bin* scheme [Mihoc et al., 2016]

Create offspring $O[l]$ **from parent** $P[l]$

$O[l] = P[l]$
randomly select parents $P[i_1]$, $P[i_2]$, $P[i_3]$, where $i_1 \neq i_2 \neq i_3 \neq i$
$n = U(0, dim)$
for $j = 0;\ j < dim \wedge U(0,1) < CR;\ j = j + 1$ **do**
  $O[l][n] = P[i_1][n] + F * (P[i_2][n] - P[i_3][n])$
  $n = (n + 1) \bmod dim$
end for

3.3.4 Optimization with adaptive Differential Evolution algorithm

Although DE is a simple and efficient algorithm, its performance depends on the control parameters (mutation and crossover probability) [Gämperle et al., 2002]. To solve the problem of finding good parameter settings some mechanisms were introduced, which, according to [Eiben et al., 1999], can be categorized in three classes: deterministic parameter control, adaptive parameter control and self-adaptive parameter control.

The JADE algorithm [Zhang and Sanderson, 2009] belongs to the second class, has an adaptive parameter control, implements a mutation strategy "$DE/curent - to - pbest$" [Zhang and Sanderson, 2009]. JADE uses also an archive of solutions, which has two functions: (i) to provide information about the progress direction; (ii) to improve the diversity in the population.

4 Experimental results

4.1 A Real-world application: Train protection systems

4.1.1 System Under Test: the TBL1+ train protection system

The system on which we ran our experiments is related to the TBL1+ system, a train protection system compatible with the European Train Control System [etc, 2008] used in Belgium and on Hong Kong's East Rail Line. Its main role is to ensure safe operation in the event of human failure. The problem specification was proposed by Siemens. We used the C code with approx. 17000 lines of code generated with the KCG code generator from the SCADE model of the system.

The system consists of a beacon on the ground that emits an electromagnetic signal. This signal is received by an antenna underneath the driving cab. As the train driver approaches a red signal, this support system switches on a light in the cab. The train driver must then confirm that they have received the warning by pressing a button. If the driver does not do this, then the emergency brake is automatically activated.

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

brake can be deactivated after 20 seconds by pressing the acknowledgment button, if the danger is no longer present.

4.1.2 From a random to a realistic environment model

We created three different Lutin models of the environment of the TBL1+ system. In all three versions, the environment model had the following main components: speed of the train, codes transmitted by the the track-side beacons, acknowledgement button for the emergency brake. We created the following three environment models:

1. **Random environment model**: This is a very simple environment model, where we assumed that we have no domain knowledge on how the TBL1+ is working. The speed of the train takes random values from one iteration to the next, and the state of the buttons (pressed or released) are also generated randomly using a uniform distribution.

2. **Realistic environment model (plain Lutin environment)**: In the second version of the environment model, we assumed that we have more domain knowledge as in the first case, and we know the possible valid values that can be transmitted by the balises. However, the codes are still picked randomly with uniform distribution.

3. **Realistic environment model fine-tuned with DE/JADE**: The third environment model is where the DE- and JADE-driven optimization was added.

4.2 Results of the DE-driven optimization

We ran DE with two different parameter settings. These settings are summarized in Table 1. Based on [Liu and Lampinen, 2002], for some problems a smaller value of F is a good setting, therefore we used a small value for F for both Setting 1 and 2. Ten independent runs were conducted for both of the settings. Average results and standard deviation are described in Table 4, as well as the results obtained with the plain Lutin environment, where no weights were added.

Parameter	Setting 1	Setting 2
Pop size	50	50
Scaling factor F	0.1	0.25
Crossover rate (CR)	0.9	0.75

Table 1: DE parameter settings.

Parameters	Setting 1 $avg. \pm stdev.$	Setting 2 $avg. \pm stdev.$	Plain Lutin environment
3	70.03 ± 0.45	70.51 ± 0.32	62.70
4	69.86 ± 0.36	70.07 ± 0.45	62.70
5	70.51 ± 0.40	70.72 ± 0.12	62.70
6	71.10 ± 0.24	71.28 ± 0.23	62.70
7	71.52 ± 1.17	72.74 ± 0.22	62.70
8	69.75 ± 0.68	70.17 ± 0.25	62.70

Table 2: MTC MC/DC coverage rates (average results and standard deviation) obtained with the DE-optimized and the plain Lutin environment. The different settings of the DE are described in table 1. The number of parameters refers to the number of weights optimized in the Lutin code.

Parameters	Setting 1 $avg. \pm stdev.$	Random environment	Plain Lutin environment
3	75.48 ± 0.00	21.29	66.45
4	75.48 ± 0.00	21.29	66.45
5	76.06 ± 0.50	21.29	66.45
6	76.16 ± 0.47	21.29	66.45
7	79.39 ± 0.47	21.29	66.45

Table 3: MTC decision coverage rates (average results and standard deviation) obtained with the DE-optimized and the plain Lutin environment. The different settings of the DE are described in table 1. The different settings of the DE are described in table 1. The detailed description of the environment models can be found in section 4.1.2. The number of parameters refers to the number of weights optimized in the Lutin code.

4.3 Results of the JADE-driven optimization

In the case of the JADE-optimized test input generation, we used the same Lutin environment model as for the experiments with DE. We used recommended parameters for JADE: 0.05 for p (determines the greediness of the mutation strategy) and 0.1 for c (controls the rate of parameter adaptation). We conducted five independent runs for each problem. Figure 8 depicts the evolution of cover rates in 108 iterations. The mean and the standard deviation is presented.

For comparison, we set Lutin's choice operator with default weights (different variables have uniform distribution in each iteration). Table 4 present obtained results for the JADE algorithm, and for the default values. As we expected, JADE has significantly better results.

Comparing the obtained results with [Szenkovits et al., 2016] we have similarities, but in this case, it was not necessary the parameter tuning, like in a "traditional" DE algorithm.

Parameters	JADE $avg. \pm stdev.$	Plain Lutin environment
5	71.82 ± 0.00	62.70
6	71.43 ± 0.60	62.70
7	71.27 ± 0.00	62.70

Table 4: MTC MC/DC coverage rates (average results and standard deviation) obtained with the DE-optimized and the plain Lutin environment. The number of parameters refers to the number of weights optimized in the Lutin code.

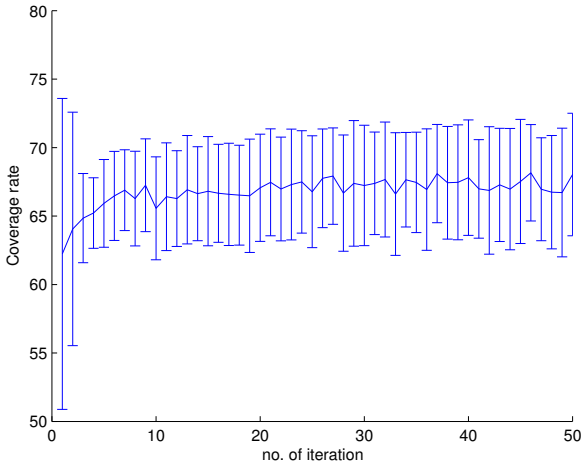

Figure 8: Population mean and standard deviation within a JADE run, for 7 parameters

5 Conclusion and future work

In this thesis, we proposed an environment-model based testing approach for reactive systems. We used the synchronous language Lutin to create an executable, non-deterministic environment model for the SUT. Because the execution environment of a reactive system is often underspecified or may change a lot, the non-deterministic nature of Lutin makes it suitable to model realistic environments. Creating an effective environment model however, often requires accurate domain-specific knowledge.

We proposed an approach in which we let different intelligent algorithms to compute the parameters of the Lutin environment model and refine the guided random exploration of the environment's state space. More specifically, we used two evolutionary algorithms:

DE and JADE. Here, we used the parameters of the Lutin environment to create the members of the population. At the end of each series of iterations of a given length, the best individuals of the population were selected and the environment models associated to these individuals were updated accordingly. The fitness functions used were the MC/DC and branch coverages of the SUT. We tested both DE and JADE on the TBL1+ system, a train protection system. We first measured the coverage rate obtained with the *plain* Lutin environment, where no optimization was added, then compared it to the results obtained with the DE- and JADE-optimized environments. In both of the cases, the coverage rate of the SUT was improved in average with 9%, without adding further domain expert knowledge. Since we are talking about a real world, fairly complex system, we can consider these results significant. Although improvements could be achieved with both DE and JADE, JADE has one major benefit over DE: JADE computes its control parameters adaptively, while in DE, we chose these parameters with a trial and error method.

As part of the future work, we aim to further increase the MC/DC coverage and to get as close to 100% as possible. We propose to try other evolutionary techniques too.

Achieving a high coverage is especially important for safety-critical systems as the TBL1+ system, where different standards regulate the process of testing. Ideally, the models should be verified with static analysis methods. In reality however, it is not always possible to verify everything statically, so dynamic verification like testing can be successfully used in combination with static analysis. Because in the testing framework proposed, the testing process is fully automated with Lutin and different intelligent methods, the guided random exploration can be done nightly, and hopefully trigger a lot of (corner) cases.

References

Official github repository of the open etcs project. https://github.com/openETCS, 2008. [Online; accessed November-2015].

J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521880386, 9780521880381.

B. Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990. ISBN 0-442-20672-0.

S. Das and P. N. Suganthan. Differential evolution: a survey of the state-of-the-art. *Evolutionary Computation, IEEE Transactions on*, 15(1):4–31, 2011.

A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 3(2):124–141, 1999.

J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005. ISBN 1852338814.

R. Gämperle, S. D. Müller, and P. Koumoutsakos. A parameter study for differential evolution. *Advances in intelligent systems, fuzzy systems, evolutionary computation*, 10:293–298, 2002.

D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-15181-8. URL http://dl.acm.org/citation.cfm?id=101969.101990.

I. Inza, P. Larranaga, R. Etxeberria, and B. Sierra. Feature subset selection by bayesian network-based optimization. 1999.

E. Jahier. The lurette v2 user guide. Technical report, Verimag Research Report, 2004.

C. B. Jones. *Systematic Software Development Using VDM (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. ISBN 0-13-880733-7.

G. T. Leavens and Y. Cheon. The java modeling language (jml) home page. http://www.eecs.ucf.edu/~leavens/JML//index.shtml. [Online; accessed May-2017].

J. Liu and J. Lampinen. On setting the control parameter of the differential evolution method. In *Proc. 8th Int. Conf. Soft Computing MENDEL 2002*, pages 11–18, 2002.

T. D. Mihoc, R. I. Lung, N. Gaskó, and M. Suciu. Approximation of (k,t)-robust equilibria. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 805–811, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4206-3. doi: 10.1145/2908812.2908877. URL http://doi.acm.org/10.1145/2908812.2908877.

P. Raymond, Y. Roux, and E. Jahier. Lutin: A language for specifying and executing reactive scenarios. *EURASIP J. Emb. Sys.*, 2008, 2008.

R. Sagarna, J. A. Lozano, and P. M. Lardiazabal. On the performance of estimation of distribution algorithms applied to software testing. 2003.

R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

A. Szenkovits, N. Gaskó, and E. Jahier. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*, chapter Environment-Model Based Testing with Differential Evolution in an Industrial Setting, pages 819–830. Springer International Publishing, Cham, 2016. ISBN 978-3-319-31204-0. doi: 10.1007/978-3-319-31204-0_52. URL http://dx.doi.org/10.1007/978-3-319-31204-0_52.

M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123725011,

9780080466484.

M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, Aug. 2012. ISSN 0960-0833. doi: 10.1002/stvr.456. URL http://dx.doi.org/10.1002/stvr.456.

J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003. ISBN 0321179366.

J. Zhang and A. C. Sanderson. Jade: adaptive differential evolution with optional external archive. *Evolutionary Computation, IEEE Transactions on*, 13(5):945–958, 2009.