Model-based Testing for Reactive Systems with Intelligent Methods: State of the Art and Challenges

Annamária Szenkovits[1]

Cluj-Napoca Faculty of Mathematics and Informatics, Babeş-Bolyai University, Kogalniceanu 1, 400084 Cluj-Napoca, Romania

Abstract. 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. Model-based testing is a testing approach that can facilitate the automatic test-case generation and thus testing costs can be significantly reduced.
The goal of this article is to address some of the fundamental problems of automatic test-case generation in safety critical, reactive systems. The research involved also focuses on the development and analysis of intelligent methods for the optimization of the automatic test-case generation process. Some of the main areas of interest are: statistical testing, evolutionary testing and estimation of distribution algorithms used in test-automation.
The practical part of the thesis aims to test the proposed methods and algorithms on problems within the domain of railway automation.

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 [Bei90].

Model-based testing is a testing approach that can facilitate automatic test-case generation. Contrary to traditional testing, instead of defining individual test cases, the model-based approach first constructs the behavior model or usage model of the system and test cases are generated automatically or semi-automatically based on the model. As a consequence, testing costs can be significantly reduced.

The report is structured as follows. Section 2 briefly describes the process of model based testing, presents its benefits, as well as the main modeling notations and test selection criteria. Section 3.1 provides an introduction to the area where the methods and algorithms suggested during the research process will be tested. This area is namely the area of reactive systems, more precisely, the domain of train control and protection systems. Section 4 presents the main elements of the research process. Part 4.1 focuses on the fundamental literature relevant for the research areas related to this thesis, namely: Software Testing, Model Based Testing, Formal Methods, Genetic Algorithms, Artificial Intelligence, Railway

Operation and Control, and Software Engineering. Finally, section 4.2 describes the main directions and domains that will be investigated during the research process, while

2 Foundations of model based testing

In this section, 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 main modeling notations and test selection criteria are described.

2.1 The process of model based testing

The main idea of the 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. Model the system under test (SUT) and/or its environment.
2. Generate abstract tests from the model.
3. Concretize the abstract tests to make them executable.
4. Execute the tests on the SUT and assign verdicts.
5. Analyze the test results. [UL07]

Thus, the first step is building a model of the SUT and/or its 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 given functionality of the SUT (requirements-based selection criteria), it can relate to the structure of the model (state coverage, transition coverage), or to stochastic characterisations (randomness or user profiles). During the third step, the test selection criteria are transformed into test case specifications which formalize the test selection criteria. 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.

When designing the model, the question arises, whether the testers should reuse the model that was used to develop the system, or whether they should build a completely new model from scratch. In practice, there are many possibilities, however, based on [UL07], it is advisable to avoid the two extreme cases and choose a middle way instead. Reusing the development model usually comes with the disadvantage that the test cases and the system itself are not independent enough. For example, if there is a fault in the model itself, it will propagate to both the system and test cases. In addition, the model used for development is usually too detailed for test generation, and several different aspects should be left out from it. The other extreme, i.e., to generate a new model from scratch can be extremely useful, especially when it is produced by a testing team different from the developing team. This way, the independence between test cases and the system can be maximized. Although very effective, this approach can be also very expensive. The middle way thus would mean to reuse some part of the developing model, and than adapt it in order to facilitate test generation. For example, it is common to reuse some of the high-level class diagrams and use-cases, and then to add the behavioral details needed for test generation. According to [UL07], the typical simplification steps of the development model are the following:

– Focus primarily on the SUT
– Show only those classes (or subsystems) associated with the SUT and whose values will be needed in the test data
– Include only those operations that you wish to test
– Include only the data fields that are useful for modeling the behavior of the operations that will be tested
– Replace a complex data field, or a class, by a simple enumeration. This allows you to limit the test data to several carefully chosen example values (one for each value of the enumeration).

2.2 Benefits of model based testing

One of the main benefits of model based testing relies in its **fault detecting** abilities. Since test cases are usually derived directly from functional requirements, it is more likely to detect faults in the SUT. Comparative studies presented in [UL07] show that the number of faults detected in the SUT have always been greater or equal to the number of faults detected with manual testing. However, the effectiveness of model based testing in fault detection still depends on the ability of the tester to design a good model and to apply the right test selection criteria.

Another benefit of the model based testing process is its role played in **reducing testing costs and time**. Instead of defining individual test cases like in the classical testing methods, a model of the software behavior is created and specific test cases are derived from this model and are applied to the SUT. The generation of the test cases can be done automatically or semi-automatically, this way the costs related to testing can be significantly reduced [UPL12]. Usually, manual testing has lower initial design costs, but its ongoing execution costs make it expensive as the number of releases increases. Automated test scripts based on a model on the other hand, are generally more expensive to design initially but can become cheaper than manual testing after several releases.

Model based testing can also **improve test quality**, mainly because the test cases are easy to relate to the original system requirements. Measuring the quality of the test suite becomes easier in the sense that it is possible to quantify the coverage of the model. Also, by using automated test generator tools, the design process becomes reproducible. In addition, the tester can produce many more tests than with manual test design.

System requirements are usually informal and are formulated in natural language. Therefore, they may often contain unclear specifications, omissions, or contradictions. Model based testing can help in finding these kinds of **requirements issues**, since the consistency and completeness of the requirements are checked. Because the model is usually expressed in a domain-specific modeling language, it becomes easier to understand by the domain experts. Also, since the process encourages early modeling, the requirements issues are more likely to be found in the requirements gathering phase rather than in later phases (e.g. design and implementation). Faults are much cheaper to fix in the early phases.

When measuring test quality, it is important that each test case can be easily related to the model, to the test selection criteria, and also to informal system requirements. This ability is called **traceability**. Since the test cases are directly derived from the model, model based testing obviously facilitates traceability.

2.3 Notations for modeling

There are several abstraction techniques and notations for model based testing available. According to [UL07], these notations can be grouped as follows.

- **Pre/post (or state based) notations**: The system is modelled 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 B[1], the UML Object Constraint Language (OCL)[2], the Java Modeling Language (JML)[3], VDM[4] 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.

[1] B method home page: http://www.methode-b.com/en/; last visited on: 18/05/2014
[2] OMG OCL specification page: http://www.omg.org/spec/; last visited on: 18/05/2014
[3] JML home page: http://www.eecs.ucf.edu/ leavens/JML//index.shtml; last visited on: 18/05/2014
[4] VDM home page: http://www.vdmportal.org/twiki/bin/view; last visited on: 18/05/2014

- **Operational notations**: When using operational notations, the system is modelled 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. As a consequence, statistical notations are typically used together with other notation types. 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 [5] for the modeling of continuous systems use data flow notations. The language Lustre will be presented in section 4.3.

2.4 Test selection criteria

[UL07] introduces a family of coverage criteria that measure how well the generated test suite covers the model. These coverage criteria remain independent from the code of the SUT. As a result, it is possible to start designing the test cases even before the development of the code has been started. However, in practice it is advisable to combine model based coverage criteria with code based criteria, after the code of the SUT has become available.

This following section offers a brief overview of the model based criteria. For a complete description please refer to [UL07]

Structural model coverage criteria Structural model coverage criteria originate from code based coverage criteria [AO08]. They can be further divided into several categories.

- **Control-flow-oriented coverage criteria**: The most relevant of them are the following.
  - **Statement coverage**: The test suite must execute every reachable statement.
  - **Decision coverage (or branch coverage)**: the test suite must ensure that each reachable decision is made true by some tests and false by other tests.
  - **Path coverage**: The test suite must execute every satisfiable path through the control-flow graph.

[5] Simulink home page: http://www.mathworks.com/products/simulink/; last visited on: 17/05/2014

- **Data-flow-oriented coverage criteria**: Control flow graphs can be annotated with *definition* and *use* of variables. The graph then becomes a data-flow graph. Informally, the *definition* of a given variable is writing to the variable, while the *use* of a variable is reading from it. Given these definitions, the aim of data-flow-oriented coverage criteria is to test all definitions, all uses, and all definition-use paths.
- **Transition-based coverage criteria**: Transition-based coverage criteria are structural model coverage criteria developed for transition based modeling notations such as finite state machines, extended finite state machines, labeled transition systems, and state charts (ref elozo fejezet). The most commonly used criteria are the following.
  - **All-states coverage**: Every state of the model is visited at least once.
  - **All-transitions coverage**: Every transition of the model must be traversed at least once.
- **UML-based coverage criteria**: The coverage criteria presented so far can be applied to UML notations, too. In addition, several UML-specific coverage criteria can be defined. For example, for **class diagrams** we can measure the following coverage measures:
  - **Generalization coverage**: This criterion requires that for every specialization defined in a generalization relationship, an instance of that specialization be created by the test suite.
  - **Class attribute coverage**: This criterion requires coverage of a set of attribute value combinations for each class in the class diagram.

Data coverage criteria The three widely used families of data coverage criteria are the following.

- **Boundary value testing**: Many of the faults found in the SUT are located at the frontier between two functional behaviors. The idea behind boundary value testing is to choose input values at the boundaries of the input domains.
- **Statistical data coverage**: Input data is randomly generated with a given distribution D.
- **Pairwise testing**: This criterion requires a test case for each possible combination of values of all pairs of parameters. It is based on the assumption that most defects are created as a result of no more than two test parameters (test values) being in a certain combination.

Requirements based criteria Based on [UL07], a requirement is a testable statement of some functionality that the product must have. Ensuring that each requirement will be tested is a key issue in the validation process. When applying model based testing, there are two main methods to measure requirements coverage.

1. Record the requirements as annotations inside the behavior model.
2. Formalize the requirements and use this formal description as test selection criterion when applying automated test generation based on the behavior model.

Statistical test generation methods In the test selection method presented in 2.4, statistical distributions were used to generate test inputs. In the case of statistical test generation methods however, statistical distributions are used to generate whole test cases, i.e., sequences of operation calls.

During statistical test generation, test cases are usually derived from the environment, because the environment model determines the usage patterns of the SUT. Therefore, the model in this case, is the representation of the system usage, not its behavior.

Statistical test generation is typically done using Markov chains, which describe the expected usage profile of the SUT. The test case generation is a random walkthrough of the Markov chain, where the random choice of the next transition is made using the probability distribution of the outgoing transitions [UL07].

3 Reactive systems

In the case of *transformational systems*, the relation of the input to the output is sufficient to completely characterise the behavior of the program. Such programs also have start and end points in time. Computation is performed with no reference to the operating environment.

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. They cannot be completely characterised in terms of the relation between input and output. The term *reactive system* was first introduced by David Harel and Amir Pnueli [HP85].

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. As suggested by its name, in the case of time-triggered reactive systems, one cycle of the system is performed within a given period of time. The cycles of event-driven systems on the other hand are triggered by certain events.

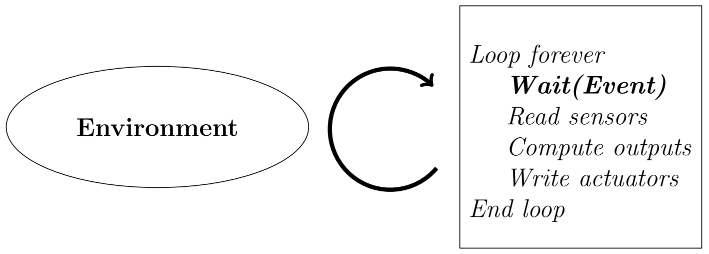
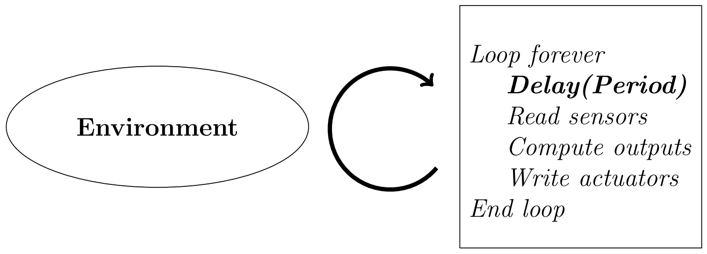


Fig. 1. Event-triggered model of a reactive system



```
Loop forever
    Delay(Period)
    Read sensors
    Compute outputs
    Write actuators
End loop
```

Fig. 2. Time-triggered model of a reactive system

3.1 Train control and protection systems

Within the domain of reactive systems, the methods and algorithms that will be designed during the research process, will be tested in the area of train control and protection systems.

The role of train control and protection systems is to ensure safe operation in the case of human failure. Thus, these systems involve technical installations that fall in the category of safety-critical systems. By definition, a system is considered safety-critical, if its failure or malfunction may result in death or serious injury to people, loss or severe damage to equipment, or environmental harm. Safety critical systems can be classified into five levels called **safety integrity levels** (SILs). The more dangerous the consequences of a software failure, the higher the software integrity level will be. Table 1 describes the different safety integrity levels.

SIL	THR per hour and per function
4	10^{-9} ¡ THR ¡ 10^{-8}
3	10^{-8} ¡ THR ¡ 10^{-7}
2	10^{-7} ¡ THR ¡ 10^{-6}
1	10^{-6} ¡ THR ¡ 10^{-5}

Table 1. Safety Integrity Levels; **THR**: Tolerable Hazard Rate

The earliest form of train protection systems were **train stops**. Train stops consist of moveable clamps, located beside the signals. The clamps touch a valve on a passing train if the signal is red and open the brake line, applying the emergency brake.

The working principle of **inductive systems** is based on coils and magnetic induction. Magnets are mounted beside the rails and on the locomotive, and data is transmitted magnetically between them. One variant of inductive systems are

PZB systems (German: Punktförmige Zugbeeinflussung, meaning Point-shaped train control). PZBs can, as suggested by their name, can only influence the train when it is passing the transmitter. Their main functions are the following.

- The **vigilance control** checks whether the driver of the railway traction vehicle has acknowledged the stop signal by pressing the corresponding button.
- **The break monitoring** supervises the process of breaking before a stop signal. On the vehicle side this is done by discrete checkpoints (in case of old systems) or by calculating a braking curve that determines if the train can stop before the next red signal. If not, the break is executed.
- The **emergency break** executed in case a stop signal has been passed.

The French alternative to the PZB is the **crocodile**. The crocodile was first introduced in the 1920s. Here, the signal aspect is represented as a positive or negative voltage.

In order to introduce a unified system in the European railways and to replace the incompatible safety systems currently used, the **European Train Control System (ETCS)** was designed. Standard trackside equipment and a standard controller within the train cab are introduced. The ETCS standards are implemented progressively, having four different levels. In its final form the standard aims to eliminate the need for lineside signals, which are difficult to follow if the train travels at high speed. All lineside information will eventually be transferred to the driver electronically. The technical specification of the ETCS is managed by the European Railway Agency (ERA) [6].

Standards, regulations European railway systems are regulated by the European Committee for Electrotechnical Standardization (CENELEC) [7]. These standards are not laws, they only express recommendations. The standards apply to any safety-related software that is used in railway control and monitoring systems, including application programming, operating systems, supporting tools and firmware.

The **EN 50126** standard refers to the railway system in general, **EN 50129** to the signalling system, while the **EN 50128** to subsystem software. The standards are not publicly available.

Table 2 shows the modeling techniques recommended by EN 50128 for different SIL levels.

4 Elements of the research process

4.1 Fundamental literature

This section presents the main areas that are related to present research work, and the relevant fundamental literature for each of these areas.

[6] ERA home page: http://www.era.europa.eu/Pages/Home.aspx; last visited on: 17/05/2014

[7] CENELEC home page: http://www.cenelec.eu/; last visited on: 17/05/2014

TECHNIQUE/MEASURE	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
Data Modeling	R	R	R	HR	HR
Data Flow Diagrams	-	R	R	HR	HR
Control Flow Diagrams	R	R	R	HR	HR
Finite State Machines or State Transition Diagrams	-	HR	HR	HR	HR
Time Petri Nets	-	R	R	HR	HR
Decision/Truth Tables	R	R	R	HR	HR

Table 2. Modeling techniques; R-recommended, HR-Highly recommended; *Source:* ***CENELEC EN 50128***

- Testing [Mye04, UL07]
- Model Based Testing [AO08, BJK$^+$05]
- Formal Methods [HU90]
- Evolutionary Computing [ES03]
- Artificial Intelligence [RN95]
- Railway Operation and Control [Pac09]
- Software Engineering [Som07]

Related Work Besides the fundamental literature presented in section 4.1, this section reviews some further works related to the intelligent methods used for model-based automatic test generation.

[PTF98] present the principles and methods of statistical software testing. Test profiles and sizes are derived based on structural and functional criteria. The method applied here is also demonstrated based on an experiment involving safety-critical systems from different application areas. In [TFW93] the method proposed for statistical testing in [PTF98] is described in more detail for functional testing.

[GDGM01] and [ADG04] introduce a generic method for path-based statistical testing based on any given graphical representation of the system's behavior. Uniform random generation routines are used for choosing test cases from the set of all possibilities.

In [BScGG06] and [BSGG07] the method described in [ADG04] is optimized for large programs, where the fraction of feasible paths in the control-flow graph is usually small compared to the unfeasible paths. A path in the control-flow graph is considered feasible if there exists an input case exerting the path. The approach introduced is called *EXIST* (*Exploration - eXploitation Inference for Software Testing*). It is a generative learning approach based on Estimation of Distribution Algorithms (EDAs) and Online Learning that aims to maximize the number of distinct feasible paths covered in a test suite.

[CCRS02] and [IKNH94] use an evolutionary algorithm to automatically generate a test program for pipelined processors by maximizing a given verification metric. In [CL11] Genetic evolutionary algorithms (GEA) are also used for test generation. The problem of parameter selection is discussed and a Markov chain based method is used to model the test generation process and to parametrise the process characteristics. The method is used here in particular for generating test cases to verify hardware design for semiconductor industry.

[Say99] discusses several software testing methods based on Markov chain usage models. In [DZ03] a framework for testing time-critical systems and software is described. The framework combines statistical usage testing based on Markov chain usage models and specification-based monitoring using sequence diagrams and formal description techniques.

Finally, [MP96] describe the theory of EDAs, while [SLL03] present the principles of evolutionary testing and the possibilities of using EDAs for testing, and compares the performance of different algorithms based on experimental results.

4.2 Topics to be investigated

This section reviews several intelligent methods applied in testing. During the research process, these methods will be studied more in detail and the possibility of improving them and adapting them for reactive systems will be investigated. Besides the intelligent methods, some other topics are presented that are relevant for testing reactive systems.

For evaluating the methods proposed, we will carry out experiments using simulations of a real-world, industrial problem.

Evolutionary testing In Evolutionary testing, heuristic combinatorial optimization techniques are used for test input generation. Examples of such techniques are: Simulated Annealing [KGV83], Tabu Search [GL97], and evolutionary algorithms.

In the class of evolutionary algorithms Genetic Algorithms (GAs) are one of the most popular and best known techniques for solving optimization problems [ILES99, SLL03]. GAs are a population based search method and they involve the following main steps:

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.

The domain of possible inputs, i.e., the possible test cases is typically too large to be exhaustively explored, even for small programs. The dimensions of the search space are directly related to the number of input parameters of the SUT [VLW$^+$13]. Since evolutionary algorithms are able to produce effective solutions for complex and poorly understood search spaces with multiple dimensions, they can also be successfully applied for testing. However, the greatest challenge remains to formulate the testing task as an optimization problem. This will influence the success of the test case design and test input generation.

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

Testing with estimation of distribution algorithms Although GAs have been applied to many problems with good results [SLL03], the determination of the parameters needed by the algorithm (crossover and mutation operators, probabilities of crossover and mutation, population size, number of generations, etc.) can be difficult and can lead itself to an optimization problem.

These problems are eliminated in the Estimation of Distribution Algorithms [LELP00]. In the case of EDAs, instead of the classical crossover and mutation operators, the probability distribution of the selected individuals is estimated and this distribution is then sampled to create the next population . Based on the experiments described in [ILES99] EDAs obtain the same quality result with fewer generations as other evolutionary algorithms.

Formally, let $\boldsymbol{X} = (X_1, X_2, ..., X_n)$ denote an n-dimensional random variable, and $\boldsymbol{x} = (x_1, x_2, ..., x_n)$ a possible instantiation of $\boldsymbol{X}$. The joint probability distribution of $\boldsymbol{X}$ is denoted by $p(\boldsymbol{x}) = p(\boldsymbol{X} = \boldsymbol{x})$. The conditional probability of X_i given the value x_j of the variable X_j is represented as $p(X_i = x_i|X_j = x_j)$ or simply as $p(x_i|x_j)$. D_l will denote the population of the l-th generation and D_l^{Se} the selected individuals. D_l^{Se} constitute a data set of N cases of $\boldsymbol{X} = (X_1, X_2, ..., X_n)$.

For each generation, the probability of an individual being among the selected individuals will be estimated based on D_l^{Se}. Formally, the joint probability distribution of the l-th generation will be calculated as $p_l(\boldsymbol{x}) = p(\boldsymbol{x}|D_{l-1}^{Se})$.

Based on these notations, the pseudocode for the abstract EDA can be written as [SLL03]:

```
$Do \leftarrow$ Generate $M$ individuals (the initial population) randomly
    Repeat for $l$=1,2,..., until stopping criterion met
    $D_{l-1}^{Se} \leftarrow$ Select $N \leq M$ individuals from $D_{l-1}$
    $p_l(\boldsymbol{x}) = p(\boldsymbol{x}|D_{l-1}^{Se}) \leftarrow$ Estimate the probability distribution of an individual being among the selected individuals
    $D_l \leftarrow$ Sample $M$ individuals (the new population) from $p_l(\boldsymbol{x})$
```

The selection step is done using the strategies in evolutionary computing, by fitness functions. The key step of the algorithm is how the probability distribution is estimated at each generation.

Online learning for statistical testing The *EXIST* (*Exploration - eXploitation Inference for Software Testing*) framework presented in [BScGG06] and

[BSGG07] proposes an adaptive sampling mechanism inspired by the Estimation of Distribution Algorithms (EDAs) and online learning for statistical software testing.

The framework generates the test cases based on the control-flow graph of the SUT. The paths in the graph can be either feasible or infeasible. A path is infeasible if it violates some dependencies between the different parts of the program. A constraint solver can label the paths as feasible or infeasible. The *EXIST* framework was developed for systems where the number of the feasible paths compared to the infeasible ones is small. It aims to retrieve distinct feasible paths with high probability.

Starting from an initial set of labeled paths, the *EXIST* iteratively generates candidate paths based on the current distribution of the program paths and updating this distribution after the path has been labeled as feasible or infeasible. The number of available labeled paths is limited by the labeling cost (because of the runtime of the constraint solver). The probabilistic model is built on top of an extended Parikh map [HU90] representation. The representation provides a propositional description of long structured sequences (program paths).

EXIST involves two modules: the **Init** module and the **Decision** module. The **Init** module estimates the probability for a path to be feasible based on its extended Parikh representation, while the **Decision** module uses the distribution of the program paths to iteratively construct the current path *s*.population

Test input generation As mentioned in section 3, reactive systems have cyclic behavior. At each cycle they read the inputs coming from their environment, compute the outputs and update the internal state of the system. Considering this, instead of generating a single test input, the tester has to provide test sequences, i.e., sequences of input vectors.

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

4.3 Tools for modeling and automated test generation for reactive systems

To investigate the way evolutionary testing and other intelligent methods are applicable for reactive systems, we propose to extend the Lutin automatic test generator tool for reactive systems with an evolutionary testing module. The Lutin tool generates the test inputs based on the Lustre descriptions of the programs. Sections 4.3 and 4.3 present the description language Lustre, and the automatic test generator tool Lutin for reactive systems.

Lustre Lustre[8] is a synchronous language[9] based on the data flow model and designed for the description and verification of reactive systems [CPHP87, HCRP91]. The language can be used for both writing programs and expressing program properties.

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 program has a cyclic behavior, so that at the nth execution cycle of the program, all the involved streams take their nth value. A node defines one or several output parameters as functions of one or several input parameters. All these parameters are streams.

Listing 1.1 shows an example[10] of a Lustre node.

Listing 1.1. Example of Lustre code

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

The node defined in listing 1.1 takes as input the Boolean stream $A = (A_1, A_2, ..., A_n, ...)$ and defines as output another Boolean stream $never\_A = (never\_A_1, never\_A_2,$
$..., never\_A_n, ...)$. The output is true if and only if the input has never been true since the beginning of the program execution.

A Lustre node can be represented as an **operator network**. An operator network depicts the way that input flows are transformed into output flows through their propagation along the program paths. The operator network equivalent to the node in listing 1.1 is illustrated in figure 3.

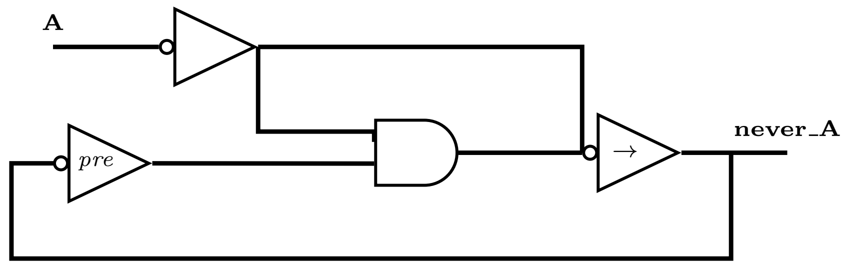


Fig. 3. The operator network for the node Never

[8] Lustre home page: http://www-verimag.imag.fr/Lustre-V6.html; last visited on: 17/02/2014

[9] a programming language optimized for programming reactive systems

[10] Example taken from http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf; last visited on: 17/02/2014

Assertions can be included into the body of a Lustre program. They are boolean expressions that should be always true. Safety properties, the properties of a program's environment can be easily specified by using the assertion me chanism.

Lutin Lutin [11] is an automatic test generator for reactive programs that focuses on functional testing. This means that the SUT will be treated as a black-box, for which we want to check some properties [Jah04].

The language is based on the use of descriptions of the environment (constraints) and the expected properties of the SUT. The main problem is to solve the constraints and to randomly generate inputs that satisfy the assertions which can be both boolean and numerical constraints [RNHW98]. In the first prototype of Lutin the environment behavior was described by Lustre specifications which described what realistic SUT inputs should be. Because Lustre was found too restrictive to express different testing scenarios, Lutin was redesigned. Now, a Lutin program is basically an automaton where each transition is associated to a set of constraints that define the possible outputs, and a weight that defines the relative probability for each transition to be taken. Listing 1.2 shows an example of Lutin code.

Listing 1.2. Example of Lutin code

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

Transitions are realized with the **choice operator** —, as illustrated in the code example 1.2. It is possible to favor one branch over the other using **weight directives** (:3 and :1 in the code example).

Improving Lutin parameters Lutin performs a guided random exploration of the SUT input state space by asking experts to write programs that describe the usage of the system. Instead of asking experts, a way to improve Lutin would be to let some evolutionary algorithms choose some parameters of Lutin programs, such as choice point weights or variable bounds (described is chapter 4.3).

5 Acknowledgement

This material is based upon work supported by the Siemens international Railway Automation Graduate School (iRAGS) and the SCADE Academic Program.

[11] Lutin home page: http://www-verimag.imag.fr/Lutin,107.html; last visited on: 17/02/2014

Bibliography

[ADG04] M.-C. Gaudel A. Denise and S.-D. Gouraud. A generic method for statistical testing. *SSRE*, pages 25–34, 2004.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[Bei90] Boris Beizer. *Software Testing Techniques (2Nd Ed.).* Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BJK$^+$05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[BScGG06] Nicolas Baskiotis, Michle Sebag, Marie claude Gaudel, and Rine Gouraud. Exist: Exploitation/exploration inference for statistical software testing. In *in On-line Trading of Exploration and Exploitation, NIPS 2006 Workshop*, 2006.

[BSGG07] Nicolas Baskiotis, Michle Sebag, Marie-Claude Gaudel, and Sandrine-Dominique Gouraud. A machine learning approach for statistical software testing. In Manuela M. Veloso, editor, *IJCAI*, pages 2274–2279, 2007.

[CCRS02] Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, and Giovanni Squillero. Evolutionary test program induction for microprocessor design verification. *2012 IEEE 21st Asian Test Symposium*, 0:368, 2002.

[CL11] Adriel Cheng and Cheng-Chew Lim. Markov modelling and parameterisation of genetic evolutionary test generations. *Journal of Global Optimization*, 51:743–751, 2011.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[DZ03] Winfried Dulz and Fenhua Zhen. Matelo - statistical usage testing by annotated sequence diagrams, markov chains and ttcn-3. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, pages 336–, Washington, DC, USA, 2003. IEEE Computer Society.

[ES03] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* SpringerVerlag, 2003.

[GDGM01] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 5–12, Nov 2001.

[GL97] Fred Glover and Manuel Laguna. *Tabu Search.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[HP85] 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.

[HU90] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

[IKNH94] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test program generation for pipelined processors. In *Computer-Aided Design, 1994., IEEE/ACM International Conference on*, pages 580–583, Nov 1994.

[ILES99] I. Inza, P. Larranaga, R. Etxeberria, and B. Sierra. Feature subset selection by bayesian network-based optimization. 1999.

[Jah04] Erwan Jahier. The lurette v2 user guide. Technical report, Verimag Research Report, 2004.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[LELP00] Pedro Larranaga, Ramon Etxeberria, Jose A. Lozano, and Jose M. Pena. Combinatorial optimization by learning and simulation of bayesian networks. In *in Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 343–352. Morgan Kaufmann, 2000.

[MP96] Heinz Mühlenbein and Gerhard Paass. From recombination of genes to the estimation of distributions i. binary parameters. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, PPSN IV, pages 178–187, London, UK, UK, 1996. Springer-Verlag.

[Mye04] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, 2 edition, 2004.

[Pac09] Jrn Pachl. *Railway Operation and Control.* VTD Rail Publishing, 2 edition, 2009.

[PTF98] H. Waeselynck P. Thevenod-Fosse. Software statistical testing based on structural and functional criteria. 1998.

[RN95] Stuart Russel and Peter Norvig. *Artificial Intelligence - A Modern Approach.* Prentice-Hall, Englewood Cliffs, 1995.

[RNHW98] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 200–209, Dec 1998.

[Say99] Kirk D. Sayre. *Improved Techniques for Software Testing Based on Markov Chain Usage Models.* PhD thesis, 1999. AAI9973502.

[SLL03] Ramn Sagarna, Jos A. Lozano, and Paseo Manuel Lardiazabal. On the performance of estimation of distribution algorithms applied to software testing. 2003.

[Som07] Ian Sommerville. *Software Engineering.* Pearson Education, 9 edition, 2007.

[TFW93] Pascale Thevenod-Fosse and Hlne Waeselynck. On functional statistical testing designed from software behavior models. In CarlE. Landwehr, Brian Randell, and Luca Simoncini, editors, *Dependable Computing for Critical Applications 3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 3–28. Springer Vienna, 1993.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

[VLW$^+$13] Tanja E. Vos, Felix F. Lindlar, Benjamin Wilmes, Andreas Windisch, Arthur I. Baars, Peter M. Kruse, Hamilton Gross, and Joachim Wegener. Evolutionary functional black-box testing in an industrial setting. *Software Quality Control*, 21(2):259–288, June 2013.