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

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

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

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

I. INTRODUCTION

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

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

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

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

A. Related Work

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

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

II. FUNCTIONAL TESTING OF REACTIVE SYSTEMS

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

 ISBN: 978-1-61208-370-4

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

Figure 1. Example of Lustre code.

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

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

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

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

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

A. Describing the SUT properties

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

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

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

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

B. Modeling the environment

Due to the reactive nature of the SUT it is necessary to have a model of the environment. This way we can generate test sequences without actually running the SUT in its real environment. There are specialized tools for describing the environment of reactive systems. Since we are testing programs written in Lustre and SCADE, in our work, we will use Lutin [5] (a language derived from Lustre) to model the environment.

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

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

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

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

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

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

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

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

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

 ISBN: 978-1-61208-370-4

C. Optimizing the environment model

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

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

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

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

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

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

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

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

III. Experimental validation

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

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

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

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

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

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

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

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

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

 ISBN: 978-1-61208-370-4

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

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

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

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

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

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

IV. Conclusion and Future Work

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

References

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

[2] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," Proceedings of the IEEE, vol. 79, no. 9, Sep 1991, pp. 1305–1320.

[3] The lustre v6 reference manual. [Online]. Available: http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf [retrieved: august, 2014]

[4] F. X. Dormoy, "Scade 6 a model based solution for safety critical software development," ERTS 2008, 2013.

[5] P. Raymond, Y. Roux, and E. Jahier, "Lutin: A language for specifying and executing reactive scenarios." EURASIP J. Emb. Sys., vol. 2008, 2008.

[6] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[7] T. E. Vos et al., "Evolutionary functional black-box testing in an industrial setting," Software Quality Control, vol. 21, no. 2, Jun. 2013, pp. 259–288.

[8] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Evolutionary test program induction for microprocessor design verification," 2012 IEEE 21st Asian Test Symposium, 2002, p. 368.

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

[10] A. Cheng and C.-C. Lim, "Markov modelling and parameterisation of genetic evolutionary test generations," Journal of Global Optimization, vol. 51, 2011, pp. 743–751.

[11] L. d. Bousquet and N. Zuanon, "An overview of lutess: A specification-based tool for testing synchronous software," in Proceedings of the 14th IEEE International Conference on Automated Software Engineering, ser. ASE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 208–215.

[12] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon, "Lutess: A specification-driven testing environment for synchronous software," in Proceedings of the 21st International Conference on Software Engineering, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 267–276.

[13] B. Marre and A. Arnould, "Test sequences generation from lustre descriptions: Gatel," in Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ser. ASE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 229–237.

[14] B. Marre and B. Blanc, "Test selection strategies for lustre descriptions in gatel," Electronic Notes in Theoretical Computer Science, vol. 111, Jan 2005, pp. 93–111.

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

[16] E. Jahier, S. Djoko-Djoko, C. Maiza, and E. Lafont, "Environment-model based testing of control systems: Case studies," in Tools and Algorithms for the Construction and Analysis of Systems, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds. Springer Berlin Heidelberg, 2014, vol. 8413, pp. 636–650.

[17] E. Jahier, P. Raymond, and P. Baufreton, "Case studies with lurette v2," Int. J. Softw. Tools Technol. Transf., vol. 8, no. 6, Oct. 2006, pp. 517–530.

 ISBN: 978-1-61208-370-4