Proceedings of 4th Spring/Summer Young Researchers' Colloquium on Software Engineering
SYRCoSE 2010, Nizhny Novgorod, p. 37 – 42

A GA-based approach for test generation for automata-based programs

Andrey Zakonov, Oleg Stepanov (research supervisor), Anatoly Shalyto (research supervisor)
Fac. of Information Technologies and Programming
St. Petersburg State University of Information Technologies, Mechanics and Optics
Saint-Petersburg, Russia
e-mail: andrew.zakonov@gmail.com, oleg.stepanov@gmail.com, shalyto@mail.ifmo.ru

Abstract—**Automata-based approach is often used for developing complex systems. Model Checking is commonly used to check conformance of the system against its specification. However, verification techniques don't allow checking the system in whole, as system consists of not only the model, but also control objects, which are not suitable for model checking. In this paper we propose an approach for testing of automata-based programs. We use EFSM and contracts to extend model with specification requirements and we demonstrate how genetic algorithms could be used to automate generation of tests to find faults in the system in whole.**

Keywords- Automata; EFSM; Testing; Genetic algorithms

I. INTRODUCTION

Automata-based program consists of a finite state machine or any other (often more complicated) formal automata and number control objects, which the model interacts with [1]. The most commonly used technique for verifying automata-based programs is Model Checking [2] because it can be used with very high degree of automation. However Model Checking suits only for verification of the automata, but not the system in whole. Controlled objects behavior and interaction of the automata and their controlled objects are not checked in this approach. Therefore there could be undetected errors left in the automata-based system, even if the automata itself was successfully verified against its specification.

In this paper we propose to use testing to check the automata-based system in whole. Software testing is normally a labor intensive and very expensive task. It accounts for about half of a typical software project life cycle [3]. This means that straightforward approach to testing, such as manual testing, is not the best option. Recently there has been much interest in automated test data generation [4]. Even though testing cannot guarantee the correctness of a program, large number of tests does contribute significantly to the identification and reduction of faults, improving the likelihood that the software implementation will succeed. Therefore this paper includes description of an approach for testing automata-based programs and a way to automate this process using genetic algorithms.

We propose to use testing to check implementation conformance against its specification. Specification given in natural language is suitable only for manual testing. In order to automate testing process, specification must be presented in some formal way. In our approach we include as much specification as it's possible in the automaton, so it would contain the instruments of its own verification. Finite state machines (FSMs) are commonly used for the purpose of automata description. However, a FSM can only model simple reaction of the system to its input events; variables and guard conditions on transitions are needed in order to model a system with complex behavior and data dependencies. Using extended finite state machines (EFSMs), which support variables and guard conditions, is a reasonable choice to describe some of the specification requirements in the automata. As it was proposed in [5] we use contracts [6] to include even more specification requirements in the automata. Having specification requirements included in the program makes it possible to automate checking of these requirements while test is executed. Moreover requirements can be used to aim test generation at detecting situations, when they are not fulfilled.

There is one more reason to use EFSMs. Most programs are designed to interact with some environment: program receives events and input data; automata react to these events and produce some output data. In automata-based programs one uses controlled objects for this purpose: they receive events and provide input data, which can be used in automata in guard conditions on transitions or as other control object functions' arguments. In EFSMs such data are represented as variables. Variable in EFSM could be internal, defined inside the EFSM itself, or external – received from the control object. During testing values of external variables can be provided by the test script.

Considering the proposed description of model and its specification, we defined a test for automata program as a sequence of events and a set of external variables, which lead to specific sequence of transitions (*transition path*) of the automata. As opposed to the traditional approach, where test is a program code, we propose to describe automata test as a transition path, which is much closer to specification level and helps to shorten the gap between the specification and the implementation. Transition path which is interesting for test creation can be easily obtained from natural language specification, but to create test code we need to find

sequence of events and set of variables, that would lead to the given path execution. Obtaining sequence of events for the path is straightforward. However set of external variables is not so easy to guess: one need to find set of values, which would satisfy all transition guards on the given transition path. We propose to apply genetic algorithms to find suitable values for external variables.

Overall, this paper addresses number of problems:

- propose an approach for testing automata-based programs;
- automate test creation by providing a tool, which finds suitable sequence of events and set of external variables for a given transition path and generates test code;
- automate validation of specification requirements, included in the automata, while executing tests;
- attempt to generate tests that lead to violation of specification requirements and so reveal faults in implementation.

The rest of the paper is organized as follows. Section II gives details on proposed approach for testing automata-based programs. Section III describes genetic algorithm applied to find external variables' values. Section IV tells about proof-of-concept tool being developed and preliminary results; Section V concludes.

II. TESTING FOR AUTOMATA-BASED PROGRAMS

The following approach for developing automata-based programs and creating test suites is proposed in this paper:

1) During development include signifigant part of natural language specification in the automata, using EFSM variables, transition guards and contracts.

Controlled objects also have specification and requirements for their inputs/outputs and interaction with the automata. All this specification requirements must be fulfilled during tests execution. Benefit of having controlled object specification included in the automata is that actual implementation of this controlled object becomes less significant for testing. Given the requirements for the object's output, we can check, that automaton reacts well for any data that fulfils given requirements. And vice versa it's acceptable if program fails for the data, which don't fulfill the object's specification.

In our approach we use JML specification language [9] to enrich automata with specification requirements. JML is a design by contract approach and contracts in JML include preconditions, postconditions, and invariants. In our case, such contracts can be defined for automata states and transitions.

2) From natural language specification select interesting scenarious for testing and present them as a sequence of transitions in the automata.

We consider sequence of automata transitions (transition path in the automata) to be a convenient way to describe a test scenario, as this representation of test could be easily derived from a natural language description of a test scenario.

There is number of researches available [7], [8] that addresses the problem of finding transition paths in EFSM to achieve selected coverage criteria (e.g. state or transition coverage in the EFSM). Such techniques can be successfully used together with manual test paths selection and, combined with the approach presented in this paper, could help to automate producing of valuable test suites.

3) Find sequence of events and values of external variables, which would make automata program to execute the disired transition path.

Automaton reacts to the events and perform transitions depending on the values of external variables used in transition guards. Representation of a test as a sequence of events and values of external variables is convinient to programmatically generate test code, but it has very little sense for a developer who works with specification defined in natural language. In our approach developer can describe test scenario in natural language first and then write it down in automata terms as a sequence of transitions, which is straightforward.

We propose an algorithm to automate search for the corresponding sequence of events and set external variables to execute given transition path. There are number of requirements that these variables must meet. First of all the guard conditions on the specified transitions should be carried out. In the second place, all the control object requirements should be fulfilled, because in production use these external variables would be obtained from control objects with given specifications. Optimization algorithms have proven to be efficient for such class of problems [4]. We apply genetic algorithms to solve this search problem. Details on genetic algorithm are described in Section III.

4) Execute generated tests and check filfullment of specification requirements for this tests exectuion.

Test code, which can execute the desired sequence of transitions is useful to perform a runtime check of all the contracts included in the automata. Support of contracts is enough to include most of the specification requirements in the automata and to check them during the tests execution. Specifications written in JML, which we use as contracts in our approach, are annotations for Java code and there are number of tools [10] that are designed to check JML contracts in the runtime or for static check.

Tests that fulfill all the specification requirements doesn't reveal any errors in the program, but still are useful for regression and stress tests. However it is much more important to generate tests, which fail any of the specification requirements for the correct set of external variables and therefore reveal inconsistency between implementation and given specification.

5) Try to find set of external variables which filfulls all guards and control object requirements and fails specification requirements of the program.

To obtain such values we also use genetic algorithm with more sophisticated fitness function, which takes into account not only transition guards and control object requirements, but also all the specification contracts defined for the given path in the automata.

III. AUTOMATIZATION OF TEST DATA GENERATION

A. Optimization problem

Set of external variables can be represented as a vector of values <$x_1, x_2, \ldots, x_n$>, where x_i is an external variable, and n is number of external variables required for this transition path. Fitness function takes this vector as an argument and returns fitness value for an external variables set. The smaller fitness value is the better the proposed vector suits the given transition path. From this point of view task can be considered as a minimization problem, where we look for the set of variables with the minimum fitness value.

B. Candidate encoding

Candidate is a vector of values, as defined above. We use one-point crossover operator, which operates by choosing a random position in the vector, and then new candidate is composed of first candidate's sub-vector before that position and second candidate's sub-vector after that position.

Mutation operator replaces random position of the vector to a new random value.

C. Fitness function

Fitness function aims to provide metric for candidates, which tells how good is this candidate for a specified task. In our case task is to execute given sequence of transitions in the automaton. There is no unambiguous answer for the question of what fitness function to choose.

Approaches for testing of structured programs propose to use such criteria as branch distance [11] for fitness calculation. A branch distance is a measure of how close a particular candidate is to executing the target branch that is missed e.g., $|A-B|$ is the branch distance for the predicate $(A > B)$. The lower $|A-B|$ is the closer is A to B and the closer the candidate is to filfulling the condition. For the filfulled condition branch distance equals zero. There are researches [11], [12] that show effectiveness of described approach for structured programs testing.

In [7] branch distance based approach is used to find input test data that can cause a feasible path in an EFSM model to be traversed. In our research we extend this approach to apply it to automa-based systems. As it was described above, we must take into account not a standalone EFSM, but an automa-based program enriched with system's and control objects' specification. Moreover we aim to find set of variables not only to execute selected path, but to filfull control objects' requirements and ideally to reveal inadequacy of implementation and specification.

To obtain variable values to execute given path there are two types of conditions that should be taken into the account:

- guard conditions on the transitions of the automaton;
- specification requirements of conrolled objects that provide external variables.

These conditions are obligatory to be filfulled. Candidate that fail any of these conditions are not appropriate for test generation, as specification doesn't require system to support such inputs. So in this case fitness function should estimate how close this particular candidate was to filfulling failed conditions.

To give an accurate estimation we examine each state and transition between states on the given path separately. Every transition has the event, which enables it and may have a guard condition and an action section. In the current implementation external variables are introduced in transitions' action sections.

Control objects' specification can be included in transition contracts: preconditions and postconditions. Precoditions verify, that automaton is in correct state to use controlled object; postconditions verify, that external variable value retrieved from the controlled object meets specification requirements.

From this point of view execution of each transition in the path is divided into three small steps:

- receive event, find transition and check guards;
- check preconditions and execute the transition;
- check transition postconditions.

Each of these steps contains coditions that can be failed. Therefore for each of these steps we calculate branch distance. Fitness value for a single transition is calculated as sum of steps' branch distances.

It's important to realize that transitions are executed sequentially. This means that to achieve second transition candidate must successfully complete first one. Therefore transitions in the beginning of the path are somehow more important then transitions in the end. This fact should be taken into the account in the fitness function calculcation. In [7], [12] transition approach level metric is introduced to handle this situation.

For more accurate fitness value we consider step approach level. In such approach each step is assigned a weight value, which depends on the step's position in the path. Last step weight is the smallest, first step weight is the greatest. Overall fitness of the candidate for the given path is calculated as sum of steps' fitness multiplied by their weights.

D. Specification requirements in fitness function

Fitness function described above is aimed to find set of variables that would make possible given path execution. More desirable is to find a candidate, which reveals an inconsistesy between implementation and specification. For this purpose we need take into consideration specification requirements of the system represented as contracts that must be filfulled during the execution. We aim to fail any of these conditions, while guards and controlled objects' requirements are filfulled.

Such task requires iterated approach, as we need to select specific transition, which conditions we want to fail. For example, if we want any of the conditions on the second transition to be failed, we need all the conditions of the first transition to be filfulled, because there may be a dependency between these conditions. For different transitions selected as target fitness function is computed differently. Generally, if k^{th} transition is a target to fail some condition, then all conditions of the transitions with indexes less then k must be fulfilled.

In attempt to fail some conditions we use branch distance turned inside out. If condition is failed then value is zero.

The closer the candidate is to failing the condition the lower the value. This reversed branch distance value is included in path fitness value calculation, similar to common step fitness, described above.

We aim to reveal faults at any transition so we iterate through the given path. At the first step we consider transition path of one transition, the first one. We perform fixed number of attempts to reveal a fault. If any found, test is generated. After fixed number of attempts we move to the next step: consider path of two transitions. We go on like this till we reach the whole given path length.

Finally, after all the iterations are done, for all revealed faults test code is generated, which can be executed separately and used for debugging and bug fixing.

IV. CASE STUDY

In this paper we present a case study that we used in our research. A proof-of-concept tool is being developed during the research. Version of the tool used for the case study contained number of limitations: only integer variable types are supported and current version is capable of providing set of variables to execute given path, but not to reveal faults.

We made up an example of specification for ATM machine and developed an automa-based system for this specification to illustrate our approach.

Sample specification of an ATM machine:

- system must perform withdrawal operations from the specified account on user requests;
- initial amount of money on the account is being retrieved from the bank at the start up. Amount must be a positive number, less or equal to 1000000;
- each time user inputs amount of money on the keyboard a transaction must be initiated. Amount must be greater then 1000 and less then 5000. If wrong input is done user must be notified about an error and operation of the system must be stopped;
- transaction must be successfully completed if after transaction there would be a positive amount of money left on the account. Otherwise transaction must be rollbacked and user must be notified about an error and operation of the system must be stopped;
- while no error occurs user can make withdrawals unlimited number of times.

For the described ATM system it is convenient to introduce number of states: initialization, user input, withdrawal operation, error in entered amount, error during the withdrawal. FSM for this system is presented on Fig. 1:

Figure 1. FSM for the ATM system.

Such model contains only basic requirements of the specification. To test such system one would need to examine specification in natural language and write tests manually.

We propose to use EFSM and to include as much specification as possible to the model. Such EFSM is presented on Fig. 2:

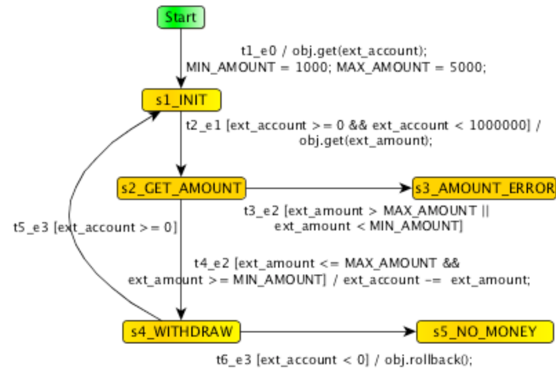

Figure 2. EFSM model of the ATM machine

Model looks more complicated this time, but on the other hand now it contains all the specification requirements, that were described in natural language. Major advantage of such representation is that now requirements are suitable to use in test generation process and for automatic checks during test executions.

Current automata-program interacts with two different control objects:

- control object responsible for bank account management. It provides amount of money on account and performs withdrawal operation;
- control object providing inputs from the user. It can be keyboard or any other device, which is not important for our purpose. Important is that this object provides an amount of money to withdraw.

Control objects' inputs are presented in model as external variables. Transition which retrieves a value from the controlled object contains following code on its label: `<object name>.get(<variable name>)`.

List of external variables with specification requirements for presented on Fig. 2 ATM model:

- `ext_account` – initial amount of money on bank account. This value is retrieved only once on the first transition. Specification requires: `0 <= ext_account < 1000000`;
- `ext_amount` – amount of money to withdraw. This variable can be retrieved unlimited number of times during the execution. Specification requires: `1000 <= ext_amount < 5000`.

Use of external variable with the defined requirements gives us ability not to depend on control object's specific implementation. Controlled objects that would be used in production are not needed for the test generation and for testing of the automata-based program. This can be critical if controlled objects are expensive or complex equipment, which are not available till the deployment of the system. Also it's important if actual controlled object implies manual input (like any keyboard does), because automatic values generation is preferable.

We considered number of different test scenarios to apply our approach. First, scenarios are defined in natural language, for example:

- user withdraws 10 times and on 11th attempt transaction fails, as not enough money on the account;
- user withdraws 5 times and on 6th attempt transaction fails, as not enough money on the account;
- user successfully withdraws 11 times;
- user withdraws 7 times and on 8th attempt incorrect amount of money is inputted.

Detailed description on how to use proposed approach for the first example follows. Test scenario should be described in terms of transactions. Scenario in terms of transition labels for the automaton given on Fig. 2:

```
t1,
t2, t4, t5, t2, t4, t5, t2, t4, t5,
t2, t4, t5, t2, t4, t5, t2, t4, t5,
t2, t4, t5, t2, t4, t5, t2, t4, t5,
t2, t4, t5, t6.
```

Sequence of transitions is given to the proof-of-concept tool as an input. Values of external variables to execute this path is produced automatically:

```
ext_acount = 28688;
ext_account1 = 3198;
ext_account2 = 4612;
ext_account3 = 2280;
ext_account4 = 2310;
ext_account5 = 4311;
ext_account6 = 1786;
ext_account7 = 3867;
ext_account8 = 1217;
ext_account9 = 2739;
ext_account10 = 519;
ext_account11 = 6376;
```

For this set of variables test code can be generated, which provides correct sequence of events and external variables values to the automata-program, so it executes actions, described in test scenario.

V. Conclusion

Simultaneously with Model Checking testing is useful to check conformance of implementation and specification while developing automata-based systems. For effective testing it is important to automate test generation process, as manual test creation is labor intensive and expensive task. In this paper we proposed an approach for testing of automata-based systems and a proof-of-concept tool demonstrating benefits of described approach. Design contracts and EFSM are used to create models containing specification requirements. Genetic algorithm is used to automate the test generation process.

We plan to provide an IDE plug-in for JetBrains MPS (Meta Programming System) [13], which has the StateMachine extension to develop automata-based programs. Seamless integration of test creation into development process would allow detecting possible implementation faults and design flaws at all development stages.

Acknowledgment

The research is conducted in scope of the Federal target program "Scientific and pedagogical personnel of innovative Russia for 2009 - 2013 years".

References

[1] A. A. Shalyto, “Logic Control and “Reactive” Systems: Algorithmization and Programming,” Automation and Remote Control, vol. 62, no. 1, pp. 1–29, 2001.

[2] E. M. Clarke, Jr. O. Grumberg and D. A. Peled, “Model Checking”, MIT Press, 1999

[3] G. Myers, The Art of Software Testing, 2 ed: John Wiley & Son. Inc, 2004.

[4] McMinn, P., “Search-based software test data generation: a survey: Research Articles,” Software Testing, Verification & Reliability, 2004. 14(2): p. 105-156.

[5] O. Stepanov, "Methods of implementation of automata-based object-oriented programs," PhD Thesis (in Russian), SPbSU ITMO, 2009

[6] B. Meyer, “Applying design by contract,” Computer, 25(10), pp. 40–51, Oct. 1992.

[7] Kalaji, A.S., R.M. Hierons, and S. Swift. “Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM),” in Software Testing, Verification, and Validation (ICST), 2009 2nd International IEEE Conference on. 2009. Denver, Colorado - USA: IEEE.

[8] Lai, R., “A survey of communication protocol testing. Journal of Systems and Software”, 2002. 62(1): p. 21-46.

[9] G. T. Leavens, A. L. Baker, C. Ruby, “Preliminary design of JML: A behavioral interface specification language for Java,” Iowa State Univ., Dept. of Comput. Sci., Tech. Rep. 98-06u, Apr. 2003.

[10] D.R.Cokand, J.R.Kiniry, “ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2,” Nijmegen Inst. for Computing and Inform. Sci., Tech. Rep. NIII-R0413, May 2004.

[11] Tracey, N., J. Clark, K. Mander, and J. McDermid. “An automated framework for structural test-data generation,” in Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on. 1998.

[12] Wegener, J., A. Baresel, and H. Sthamer, “Evolutionary test environment for automatic structural testing,” Information and Software Technology, 2001. 43(14): p. 841-854.

[13] MPS User's Guide. http://www.jetbrains.net/confluence/display/MPS/MPS+User%27s+Guide.