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

Spring/Summer Young Researchers' Colloquium on Software Engineering 2010, Nizhny Novgorod

Andrey Zakonov

Research supervisors: Oleg Stepanov, PhD
Anatoly Shalyto, PhD
Agenda

- Automata-based approach and problem of the quality assurance
- Developing and testing automata-program:
  1. Creating model and formalizing requirements
  2. Defining test scenarios
  3. Creating executable tests
  4. Running tests
- Summary
Automata-based approach

- Automata-based program consists of:
  - model, a formal automata (FSM)
  - control objects

- Model defines behavior of the system

- Control objects interact with environment (input/output)
The problem is to check program against its specification requirements.

There are three parts of automata-program that could contain errors:
- model
- controlled objects
- interaction of the automaton with its controlled objects

There are ways to check automata-model (Model Checking), but they don’t work for controlled objects and system in whole.
Proposed solution

- To use *automata-tests* to check the automata-based system in whole (model + controlled objects)

- Automata-test simulates inputs to the system and checks behavior of the system for this inputs

- Drawbacks of testing approach:
  - can not guarantee the correctness of a program
  - normally a labor intensive and very expensive task
No approach or tools to test automata-programs

Extended Finite State Machine (EFSM) related approaches don’t support an interaction with controlled objects

Traditional testing approaches can not be applied to automata-program as is:
- all benefits of automata approach would be lost
- metrics are not meaningful

Testing is labor-intensive and requires automation tools
Steps to test an automata-program

1. Formalize natural language specification
2. Describe test cases
3. Create an executable test
4. Run tests and check implementation against its specification
Agenda

- Automata-based approach and problem of the quality assurance

- Developing and testing automata-program:
  1. Creating model and formalizing requirements
  2. Defining test scenarios
  3. Creating executable tests
  4. Running tests

- Summary
I. Formalize specification

- Specification usually is described in natural language

- Example of ATM-like system:
  - system withdraws from an account
  - initially sum on account is more then 0 and less then 100000
  - user can withdraw infinitely while sum is positive
  - user enters amount to withdraw, more then 1 000 and less then 15 000
  - no more then 50 000 can be withdrawn during one day of operation

- Good only for manual testing
I. Groups of requirements

- Model’s requirements:
  - system withdraws from an account
  - user can withdraw infinitely while sum is positive
  - no more than 50,000 can be withdrawn during one day of operation

- Control objects’ requirements:
  - initially sum on account is more than 0 and less than 100,000
  - user enters amount to withdraw, more than 1,000 and less than 15,000
I. Developing a model - FSM

- We define events:
  - e0 – initialized
  - e1 – user input
  - e2 – transaction complete
  - e3 – error

- A lot of logic is hidden in control objects’ implementation
I. Covered requirements

- **Model’s requirements:**
  - system withdraws from an account
  - user can withdraw infinitely while sum is positive
  - no more than 50 000 can be withdrawn during one day of operation

- **Control objects’ requirements:**
  - initially sum on account is more than 0 and less than 100000
  - user enters amount to withdraw, more than 1 000 and less than 15 000
I. Developing a model - EFSM

- Extended Finite State Machine supports variables and suits for more complex models
I. Covered requirements

- Model’s requirements:
  - system withdraws from an account
  - user can withdraw infinitely while sum is positive
  - no more than 50,000 can be withdrawn during one day of operation

- Control objects’ requirements:
  - initially sum on account is more than 0 and less than 100,000
  - user enters amount to withdraw, more than 1,000 and less than 15,000
I. More ways to describe requirements

- Controlled objects contain some logic, as using EFSM is not always good:
  - too complex model
  - model’s requirements and control objects’ requirements would be mixed up

- Need to formalize requirements to check the model and controlled objects implementation

- Design by contract approach
  - preconditions, postconditions, invariants
I. Requirements as contracts

- Control object requirements can be added as pre- and postconditions of the transitions
- Model’s requirements can be added as invariants to the states
- Java Modeling Language (JML) to write requirements

- Benefits of such approach:
  - model shows specification requirements
  - developer-friendly syntax
I. Developing a model – EFSM+JML

- Account:
  - @ensures ext_sum >= 0
    && ext_sum <= 100000

- User input:
  - @ensures ext_x >= 1000
    && ext_x <= 15000

- Model
  - @invariant today <= 50000
I. Covered requirements

Model’s requirements:
- system withdraws from an account
- user can withdraw infinitely while sum is positive
- no more then 50 000 can be withdrawn during one day of operation

Control objects’ requirements:
- initially sum on account is more then 0 and less then 100000
- user enters amount to withdraw, more then 1 000 and less then 15 000
Agenda

- Automata-based approach and problem of the quality assurance

- Developing and testing automata-program:
  1. Creating model and formalizing requirements
  2. **Defining test scenarios**
  3. Creating executable tests
  4. Running tests

- Summary
II. Defining test cases

- Convenient to describe test scenarios in natural language
- Let’s define formally test case as a sequence of transitions in the automaton
  - easy conversion to and from natural language
  - can be generated automatically
- Test scenario looks like:
  - t1, t2, t4, t5, t2, t4, t5, t2, t4
Agenda

- Automata-based approach and problem of the quality assurance

- Developing and testing automata-program:
  1. Creating model and formalizing requirements
  2. Defining test scenarios
  3. Creating executable tests
  4. Running tests

- Summary
To execute the given path it’s necessary:

- provide events in the correct order
- provide values for the external variables

External variable values come from environment:

- no access to environment on testing stage
- automation is wanted

It’s a problem to guess these values:

- fulfill all the transition guards
- fulfill control objects’ contracts
III. Guessing variable values

- Genetic algorithm can be applied
- Fitness function estimates how good is given set of values for the desired path:
  - successful steps
  - branch distance for failed steps
  - location of failed steps
- Values with zero fitness will make the test
- GA is applied to solve optimization problem
III. GA details

- Chromosome is a vector of variable values
  - \(<x_1, x_2, \ldots, x_n>\)

- One-point crossover operator
  - \(<x_1, x_2, x_3, x_4>\) \(<x_1, x_2, x_3, y_4>\)
  - \(<y_1, y_2, y_3, y_4>\) \(<y_1, y_2, y_3, x_4>\)

- Mutation – replace random variable with random number

- Fitness function
  - branch distance: ("A >= B") = \(\begin{cases} 0, & A \geq B \\ |A - B|, & A < B \end{cases}\)
  - weighted sum, path = \(\sum_{i=0}^{m-1} f_i \times d_i\)
Example of test cases:

- **Three times** withdrawal operation is successful, forth time there is not enough on the account
- **Twenty times** withdrawal operation is successful
- Different variable values are required for these tests
III. Guessing values example (2)

- First test scenario transition path:
  - $t_1, t_2, t_3, t_2, t_3, t_2, t_3, t_2, t_4$

- Five external variables are used:
  - `ext_sum` – initial value on the account;
  - `ext_x1` – first withdrawal;
  - `ext_x2` – second withdrawal;
  - `ext_x3` – third withdrawal;
  - `ext_x4` – failed to withdraw.

- Proof-of-concept tool accepts transition path and returns set of variables
III. Generating executable tests

- Automatically found values:
  - `ext_sum = 15673;`
  - `ext_x1 = 4357; ext_x2 = 8023;`
  - `ext_x3 = 2162; ext_x4 = 9183;`

- Executable test on Java can be created and run later
- Organizing big test suits are good for regression and stress testing
Agenda

- Automata-based approach and problem of the quality assurance

- Developing and testing automata-program:
  1. Creating model and formalizing requirements
  2. Defining test scenarios
  3. Creating executable tests
  4. Running tests

- Summary
IV. Running tests

- Behavior of the system need to be checked during the evaluation of the given path
- If JML contracts are defined for states on this path they would be checked at the runtime:
  - JML Runtime Assertion Checker can be used
- In the example `@invariant today <= 50000` will be checked after each transaction
- In case of failing the condition an exception will be raised
IV. Running tests

- Implicit requirements are always checked:
  - deadlocks
  - exception
  - execution time
  - etc.

- For real control objects contracts will be useful to reveal inadequate implementation
Values that fail requirements

- Fitness function may take into the account model’s specification
- It will help to find values that fail requirements
- Examine steps of the given path sequentially:
  - try to fail at first step
  - fulfill first step and fail second
  - ...
  - fulfill first $n-1$ steps and fail $n^{th}$ step
Agenda

- Automata-based approach and problem of the quality assurance
- Developing and testing automata-program:
  1. Creating model and formalizing requirements
  2. Defining test scenarios
  3. Creating executable tests
  4. Running tests
- Summary
1. Specification is formalized using EFSMs and JML contracts
2. Test scenarios are described as a transition path
3. GA-based tool is used to find variable values for given path and executable tests are generated
4. Tests are run automatically and JML requirements fulfillment is checked at the runtime
Thank you

- Questions & Answers

- Andrew Zakonov, SPb SU ITMO
  andrew.zakonov@gmail.com