Design of Search Based Testing Infrastructure

ZL

November 26, 2010

Contents

1 Introduction **2**

2 Terminology **2**

3 Use Cases **3**
3.1 Explore all states in the given state space and find optimum . . . 3
3.2 Explore all states in the given state space and find cummulative optimum of n states 4
3.3 Search for an optimum in the given state space 4
3.4 Search for a cummulative optimum over n states in the given state space 5
3.5 Search for a cummulative optimum of n states and probabilities of their usage in mixed strategy 5
3.6 Search for an overall cummulative optimum in the given state space 6
3.7 Search for optimal combination of tests which may be used during limited time 6

4 Testing Infrastructure Overview **7**
4.1 Manager 7
4.2 Search 8
4.3 Executor 8
4.4 State Space Storage 8

5 Input Specification **9**
5.1 Set of Parameters 9
5.2 Set of Tests 9
5.3 Constraints for Search Procedure 11
5.4 Constraints for Exploring a State Procedure 11
5.5 Search Technique 11
5.6 Knowledge Obtained from Previous Exploration 11

6 Output Specification **12**
6.1 State 12

7 Related Work **12**

1 Introduction

The aim of the proposed infrastructure is to allow experimenting with applying search techniques and evolutionary algorithms in the field of program testing.

Consider situation that there are multiple tests of a given software, each has multiple parameters of different type that somehow affect test execution and you want to find best (whatever best means for you) combination of tests and their parameters with respect to a given time constraint. Moreover, if a software containing concurrency is tested, multiple executions of the same test executed with the same parameters can lead to slightly different executions because of a non-determinism of concurrent software execution. One of successful techniques for testing concurret software is noise injection which affects execution of concurrent software by adding various type of noise at specified places. Parameters of noise maker again increase number of parameters that influence test execution.

This infrastructure provides an environment where such situation can be studied. One can then experiment with different search and evolution algorithms used for finding best combinations of tests and parameters, and ask interesting questions concerning combination of tests, number of their execution, and parameters to be used in different testing strategies.

Infrastructure design presented in this document was created with respect to requirements of testing concurrent programs using IBM concurrency testing tool (ConTest) [2] and collecting coverage information from tests. This influences use cases presented in the next section and also induces some design decisions.

The document is organized as follows. In the next section, a few use cases are presented so one can get an idea what are interesting questions for the infrathatstructure. Next, a high level modular structure of the infrastructure is presented together with a high level description of functionality of each module. Then, infrastructure inputs and outputs are enumerated. Finally, a related work is very briefly commented (more less a few links which can be used as a starting point for studying related work are given).

2 Terminology

In terminology of evolutionary algorithms, tests and their parameters represent *n-dimensional state space* where number of dimensions is number of test parameters plus one dimension of tests. A *state* in such state space represents a test with specific parameters (or a test configuration). *Fitness function* is used for estimating quality of a state being explored and is represented by particular test results. A search or evolutionary algorithm is used to choose a state which should be explored in the next step of exploring the state space.

Use cases listed in the next section concern fitness function computed from a problem of coverage. *Coverage* is defined as any metric of completeness with respect to a test selection criterion. Coverage analysis consists of two steps. At first, a list of tasks to be covered is prepared. *Coverage task* is a boolean function on a test (e.g. method XY was executed). A cohesive set of coverage tasks is called *coverage model*. Then, analysis checks what tasks were covered during testing. Usually, the goal is to cover as many tasks as possible – ideally all tasks (which are feasible).

Common situation for a concurrency coverage model is that repeated execution of the same test with the same parameters gives different coverage. Therefore, there is reason to execute one test repeatedly to get more covered tasks. Lets call such execution of the same test repeatedly *pure testing strategy* and repeated execution of the same test but with different parameters *mixed strategy*. Experiments done by Shmuel Ur and his colleagues show that for a concurency coverage model and testing using noise maker there usually exist a mixed strategy where only parameters of noise maker are changing that is better than any pure strategy. One can also consider *stochastic mixed strategy* where for each test and its parameters a probability is given such that the sum of all probabilities of tests and parameters used in mixed strategy is 1. The probability represents likelihood that the next test to be executed will use those parameters.

One of considered usages of the tool is to identify tests and their parameters for regression testing. So called *regression sets* are compact sets of tests run after each code modification or at certain intervals. It is very important for these tests to be as comprehensive as possible to increase the likelihood of finding new bugs but still with respect to bounded time devoted to execution of these tests (e.g. one night). One way to create such a set of tests is to sort the tests by degree of coverage achieved. Given a fixed amount of time for running the tests, the subset with the highest degree of coverage is selected.

Since keyword *cummulative* is overloaded in the next sections, lets describe all possible usages here to avoid misunderstanding. There are three concerned dimensions over which evaluation function (e.g. set of covered tasks) can be cummulated. The first is number of executions of one test configuration (see pure testing strategy above) – called simply *cummulative value*. The second is number of different test configurations (see mixed testing strategy above) – called *cummulative value over n tests*. This value is comuputed over chosen set of n configurations. Note that these two cummulative values can be combined since we can claim cummulative value of evaluation function of a single test to be just value of evaluation function of particular test and then cummulate over multiple tests. And finally, one can also consider cummulating values over all test configurations executed during state space exploration – *overall cummulative value*.

3 Use Cases

This section describes considered use cases of the infrastructure.

3.1 Explore all states in the given state space and find optimum

There is no need for a search technique in this use case because all states in the state space must be explored. The given state space could be a representative subspace of a real state space. *Optimum* is a state that provides maximal (best) value of a given fitness function. If more states have maximal value of the given fitness function, one of them is randomly chosen.

- **Inputs:**

  - Set of tests and set of parameters
  - Constraints for exploring a state procedure

- **Outputs:**
  - State with the best value of fitness function
  - For the best state and each execution of a particular test configuration:
    * Counter identifying the execution
    * Cummulative value of fitness function
    * Cummulative set of covered tasks
    * Execution time
    * Test status (pass, fail, ...)

3.2 Explore all states in the given state space and find cummulative optimum of n states

There is no need for a search technique in this use case because all states in the state space must be explored. The given state space could be a representative subspace of a real state space. After all states in a state space are explored, further analysis identify n states that are best candidates for mixed testing strategy.

- **Inputs:**
  - Set of tests and set of parameters
  - Constraints for exploring a state procedure
  - Number of states n that gives optimal cummulative value of fitness function

- **Outputs:**
  - n states with the best cummulative value of fitness function
  - For each chosen state and each execution of particular test configuration:
    * Counter identifying the execution
    * Cummulative value of fitness function
    * Cummulative set of covered tasks
    * Execution time
    * Test status (pass, fail, ...)

3.3 Search for an optimum in the given state space

Search technique is used to choose state which is going to be explored in the next step. Searching is finished when constraints for search procedure applies.

- **Inputs:**
  - Set of tests and set of parameters

  - Constraints for exploring a state procedure
  - Constraints for search procedure

- **Outputs:**
  - State with the best value of fitness function
  - For the best state and each execution of a particular test configuration:
    * Counter identifying the execution
    * Cummulative value of fitness function
    * Cummulative set of covered tasks
    * Execution time
    * Test status (pass, fail, ...)

3.4 Search for a cummulative optimum over n states in the given state space

Search technique is used to choose state which is going to be explored in the next step. Searching is finished when constraints for search procedure applies.

- **Inputs:**
  - Set of tests and set of parameters
  - Constraints for exploring a state procedure
  - Constraints for search procedure
  - Number of states n that gives optimal cummulative value of fitness function

- **Outputs:**
  - n states with the best cummulative value of fitness function
  - For each chosen state and each execution of particular test configuration:
    * Counter identifying the execution
    * Cummulative value of fitness function
    * Cummulative set of covered tasks
    * Execution time
    * Test status (pass, fail, ...)

3.5 Search for a cummulative optimum of n states and probabilities of their usage in mixed strategy

Search technique is used to choose state which is going to be explored in the next step. Searching is finished when constraints for search procedure applies. Search finds best cummulative optimum and probabilities for a stochastic mixed strategy.

- **Inputs:**

  - Set of tests and set of parameters
  - Constraints for exploring a state procedure
  - Constraints for search procedure
  - Number of states n that gives optimal cummulative value of fitness function

- **Outputs:**
  - n states with the best cummulative value of fitness function
  - A probability for mixed strategy for each choosen state
  - For each chosen state and each execution of particular test configuration:
    * Counter identifying the execution
    * Cummulative value of fitness function
    * Cummulative set of covered tasks
    * Execution time
    * Test status (pass, fail, ...)

3.6 Search for an overall cummulative optimum in the given state space

Search technique is used to choose state which is going to be explored in the next step. Searching is finished when constraints for search procedure applies.

- **Inputs:**
  - Set of tests and set of parameters
  - Constraints for exploring a state procedure
  - Constraints for search procedure

- **Outputs:**
  - States with the best overall cummulative value of fitness function
  - For each explored state of state space and each execution of particular test configuration:
    * Counter identifying the execution
    * Cummulative value of fitness function
    * Cummulative set of covered tasks
    * Execution time
    * Test status (pass, fail, ...)

3.7 Search for optimal combination of tests which may be used during limited time

Search technique is used to choose state which is going to be explored in the next step. Searching is finished when constraints for search procedure applies. Search technique considers execution time of each state. Further analysis then choose what tests to use when limited time for testing is available (e.g. regression testing during one night).

- **Inputs:**
  - Set of tests and set of parameters
  - Constraints for exploring a state procedure
  - Constraints for search procedure
  - Time limitation for execution of chosen tests
- **Outputs:**
  - n states with the best cummulative value of fitness function
  - Expected time of running chosen tests
  - For each chosen state and each execution of particular test configuration:
    * Counter identifying the execution
    * Cummulative value of fitness function
    * Cummulative set of covered tasks
    * Execution time
    * Test status (pass, fail, ...)

4 Testing Infrastructure Overview

Infrastructure is divided into 4 cooperating modules called *Manager*, *Search*, *Executor*, and *State space storage*. Modules, their communication, and tasks are shortly described in the following subsections.

There will be also other modules of the infrastructure for instance GUI and log producer which are not described in this document since they are not involved in the main functionality of the infrastructure.

4.1 Manager

The main module called Manager controls entire execution process. At first, it processes input data (parameters, test specifications, etc.). Then, the manager initializes all other modules. The state space storage module is initialized as empty an n-dimensional matrix. Subsequently, the manager enters the testing loop which consists of the following steps:

1. Ask the search module for the next state to be explored. The state identifies a test and its parameters.
2. The chosen state is passed to the Executor which is responsible for exploring the state by executing a test with the given parameters.
3. Manager checks results and if termination conditions are not fulfilled continues with step 1.

After the testing loop, if needed, Manager initiates a further analysis of the obtained results which are stored in State space storage module (e.g. compute some interesting statistics etc.).

4.2 Search

Search module consists of an interface which is used to cooperate with the rest of the infrastructure and *pluggable search engine* which actually does search. Search module is called to provide the next state that should be explored. Search engine can via interface access the state space storage module in order to get information concerning the state space (dimensions, values of already explored states, etc.). Search module provides a list of parameters which identify test configuration to be executed in the next step.

4.3 Executor

Executor module consist of an interface which is used to cooperate with the rest of the infrastructure, *execution manager*, and several *pluggable engines*. Execution manager controls state exploration which consists of the following steps:

1. Execution of engine that does initialization of state exploration (e.g. ConTest instrumentation).
2. Execution of enginen that does initialization of test execution. This step is used to prepare test execution.
3. Execution of engine that execute particular test. Execution manager checks constraints for exploring a state and if needed interrupts execution.
4. Execution of engine that does gathering of results and computation of fitness function. Execution manager get result of fitness function and stores it in state space storage.
5. Execution of engine that does finalization of execution (e.g. cleaning after execution).
6. Execution manager checks if termination conditions of state exploration are fulfilled and if not continue with step 2.
7. Execution of engine that does finalization of state exploration (e.g. clean temporal directories).

Pluggable engines are implemented as Java classes. Currently, three types of plug-ins are considered: an empty plug-in that does nothing, a simple Java plug-in (just invocation of a method), a plug-in that runs a new Java process and waits till its end, and a plug-in that runs a new process and waits till its end.

4.4 State Space Storage

State space storage module is a service which can be used by search engine and executor to store and analyze test results. This module provides uniform storage of results and some statistical data concerning state space exploration. Results are stored in sparse n-dimensional matrices where each cell represents one state in a state space. The number of dimensions is set according to number

of parameters search engine must determine. Parameters are specified in the input file.

Storage should be attached to some analysis engine which is able to compute different statistics, summaries and views. This is an object of a future work.

5 Input Specification

Inputs are given to the infrastructure in an XML file (chosen due to its tree structure). Inputs described in this section refer to use cases listed in Section 3.

Set of parameters (include ConTest parameters) and set of tests describe state space for a search algorithm. Constraints for search procedure specify restrictions used by Manager to stop exploring state space. Constraints for exploring a state procedure specify restrictions used by Execution manager during execution of a test.

5.1 Set of Parameters

Parameter is value that influences test execution. There are several types of parameters according to place they are used. JVM parameters influences behavior of Java Virtual Machine, test parameters are passed to the main method of the test, and ConTest parameters influences ConTest behaviour (noise injection, etc.).

- Inputs:
  - For each parameter:
    - Parameter type (e.g. JVM, program, ConTest, ... – used by infrastructure and plug-ins)
    - Identifier (e.g. name of parameter)
    - Domain of parameter (one fixed value (parameter is constant), boolean, enumeration (list of possible values), integer(min, max))
    - Ordering if exist (ordinal, not ordinal)
    - Value(s) of parameter (according to domain – e.g. list of possible values, integer min, max, ...)
  - Set of parameter combinations that doesn't make sense to explore provided as a set of Java expressions (e.g. $((param1 < 0)\&\&(param2 > 100)))$ (optional, future work)
  - Set of parameter combinations that are equal – leads to the same state in the state space – provided as set of tuples (a, b) where a and b are Java expressions (e.g. $[((param1 < 0)\&\&(param2 > 100))], [(param1 > 100)\&\&(param2 < 0)]$) (optional, future work)

5.2 Set of Tests

Test is considered to be fitness function that is used to explore the chosen state.

- Inputs:
  - For each test:

        * Identifier (e.g. name)
        * Exploration intitialization plug-in and its parameters (optional)
        * Preprocessing plug-in and its parameters (optional)
        * Execution plug-in and its parameters
        * Evaluation plug-in and its parameters (returns value used as fitness function)
        * Postprocessing plug-in and its parameters (optional)
        * Exploration finalization plug-in and its parameters (optional)
        * Liveness constraints for the test (timeout, timeout of producing output)
    - Class name that will be used to represent result of the test

Initialization and finalization of exploration, preprocessing, postprocessing, and execution plug-ins are of the same type (impleplement the same interface). Currently, only three types are considered:

1. Execution of a new Java process
    - Inputs:
        - JVM executable to be used
        - JVM parameters (classpath, etc.)
        - Class representing test to be executed
        - Parameters passed to main method
2. Execution of a new process
    - Inputs:
        - Command to be executed
3. General Java class implementing appropriate interface
    - Inputs:
        - Full java class name to be used
        - List of parameters (name, value)

Evaluation plug-in is of different type (implement different interface). Currently, only one type of plug-in is considered:

1. General Java class implementing appropriate interface
    - Inputs:
        - Full java class name to be used

Class used for representing exploration result contains general results (e.g. execution time) and also analysis domain specific results (e.g. set of covered tasks).

5.3 Constraints for Search Procedure

These constraits determine when to stop searching and are checked by Manager. If no constraint is given, state space exploration continues until either the entire state space is explored or user terminates the program.

Q: Should we allow to reexplore already explored state? What happens than – the old result will be rewrited or the new result will be somehow merged with the old one?

Currently considered constraints for search procedure are:

- Inputs:
  - Timeout (optional)
  - Number of states to be explored (limits number of steps the search technique can do) (optional)

5.4 Constraints for Exploring a State Procedure

These constraints applies when a particular state in the state space is being explored and are checked by Execution manager. At least one constraint must be set.

Currently considered constraints are:

- Inputs:
  - Timeout
  - Number of iterations for each test
  - Treshold of improvement of a last executions (a, treshold)

5.5 Search Technique

Search technique is responsible for choosing the next state to be processed.

- Inputs:
  - Class that implements appropriate interface
  - Parameters for a search technique (name = value) (optional)

5.6 Knowledge Obtained from Previous Exploration

This is also part of a future work. Search technique can be provided by some interesting information concerning each state (or subset) of a state space. This information can be obtained e.g. from previous executions. Interesting information can be one of follows:

- Inputs:
  - For each state (test + parameters):
    - Expected run time
    - Previous state exploration result

6 Output Specification

Besides outputs described in this section, log file production and export of state space storage are considered. In the future, a GUI is planned to be implemented for processing outputs.

6.1 State

Considered output can be in a form of one selected state or a set of selected states. Output of a state is controlled by class representing result obtained by state exploration. The class representing result is part of input information and therefore can be influenced by user. The following example describes output of default result format of a coverage task described in Section 5.2.

- Outputs:
  - For each state (test + parameters):
    - Test identification (name)
    - List of parameters and their values used with test
    - Execution plug-in parameters (JVM, ...)
    - Cummulative value of fitness function
    - Cummulative list of covered tasks
    - Execution time
    - Test status (pass, fail, ...)

7 Related Work

There is a lot of work done in the area of *evolutionary testing* [3] and more general area of *search based software engeneering* [5]. There are research groups which focus on the combination of search techniques and testing e.g. *CREST* [1]. There is also a tool which provides similar functionality – EvoTest – product of an European Union funded research project IST-33472 [4] (the tool is still being developed).

References

[1] Centre for Research on Evolution Search & Testing
http : *//crest.dcs.kcl.ac.uk/*

[2] Concurrency Testing Tool (ConTest)
http : *//www.alphaworks.ibm.com/tech/contest/*

[3] Evolutionary Testing
http : *//www.systematic* − *testing.com/evolutionary_testing*

[4] EvoTest – Evolutionary Testing for Complex Systems
http : *//www.evotest.eu/*

[5] Repository of Publications on Search Based Software Engineering
http : *//www.sebase.org/*