TEST DATA GENERATION USING METAHEURISTICS

János OLÁH
Advisor: István MAJZIK

I. Introduction

Software testing is the process of evaluating the quality of the software under test by controlled execution, usually with the primary aim to reveal inadequate behavior or performance problems. Obviously, testing is an essential step of all software development models, however, writing tests is expensive, labor-intensive and time consuming, thus contributors often skimp on testing phase.

One of the most important task in a testing is test data generation. In software testing, test data generation is the process of identifying input data which satisfy certain criterion, e.g. coverage of selected program code. Test data generation is roughly equivalent with the simple case of test case generation, where the goal is to find an input sequence, that will drive the execution along a particular path in the control flow graph. Nevertheless, a test case is usually more than a simple input data, e.g. test case usually contain the desired output of the system under test.

Unfortunately, it's usually difficult to create realistic production data, especially early in the development stage, when discovering software bugs would be important in order to avoid expensive redesigns in later phases. Enormous effort have been made to overcome these difficulties. Through the last decades, several approaches have appeared for automatic software test data generation. In paper [1] authors divided these methods into three classes: random, path-oriented and goal-oriented automatic test data generation methods.

In this paper we present a novel automatic test data generation approach, which utilizes the idea behind model based test generation. In MBT, test cases are generated using the engineering models and formal specifications of the system under test, constructed during the planning phase (left side of figure 1 presents the general architecture of MBT). The models provide specific information for the test derivation algorithm, while formal specifications are precise, machine-readable mathematical descriptions of the expected behavior of the software under test (see [2] for details about MBT).

Test derivation is a crucial component in this approach, since the effectiveness of MBT depends on the level of automation it provides. In fully automated test case generation, models are usually translated to finite state automaton or transition system, and these are searched for possible executable paths, by model-checking, constraint satisfaction or SAT solving. An extensive survey of automated test generation is presented in [3].

II. Test data generation

In the previous section, we mentioned the classification of test data generation methods. As a remainder, test data generation is often formulated as identifying a program input which executes a given program element. Random methods obviously select input data by random selection. The advantage of these methods is the easy implementation and their generality, however in case of large state space they are very unlikely to identify proper input sequence.

Path-oriented methods reduce the test data generation to a path problem, where a path in the control flow is selected usually to trigger a selected program statement, and then the task is to generate input data to execute that path. Two popular methods in this category is symbolic execution and execution-oriented test data generation. This approach is best suited for programs with relatively small number of paths and simple control sequences, since their main disadvantage is the wastage of significant

computation effort while identifying infeasible paths in the control flow graph.

In the goal-oriented approach, the path selection is eliminated, thus the goal is to find particular input data which trigger a selected statement in the program code. Methods using this approach monitor the program execution with the current input data, and classify branches according to their influence on execution of the desired branch.

The weakness of these approaches is the program analysis stage. Both path- and goal-oriented approaches require the analysis of program code, which can be complicated in case of large program code, partly implemented program with component stubs or legacy code. Furthermore, all introduced approaches handle complex data structures with difficulty, though most modern software application deal with large and complex input data.

III. Metamodel-based test data generation

Besides engineering models describing the structure and dynamic behavior of the system under test, complex input data of the program is usually modeled by the architect. Our proposed approach for test data generation uses this metamodel, in order to generate feasible test data. The metamodel describes the abstract syntax of a modeling language, thus in our case it defines the well-formedness rules of program data, and test data generation task can be interpreted as instance model generation.

This idea is inspired by model-based test generation, but instead of engineering models we use the metamodel of test data. However, this approach only suitable when goal of testing can be defined without engineering models. The architecture of the approach is shown on the right side of figure 1.

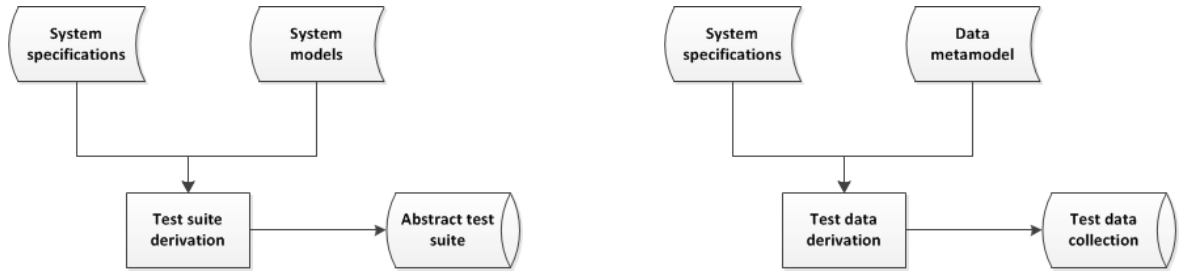

Figure 1: Left side of the figure presents traditional architecture of model-based software test generation, while the architecture of the proposed metamodel-based test data generation is shown on the right.

As it was already observed in section I., test derivation algorithm in MBT usually works on the transformed automatons of the original models. In this approach we propose the utilization of metaheuristics for test data generation.

A. *Metaheuristics*

Metaheuristics is the primary subfield of stochastic optimization applied for a very wide range of problems. However the term is somewhat misleading, optimization methods in this category apply some degree of randomness to find an optimal solution. Metaheuristics are advantageous in problems described as “I know when I see it”. In these problems there isn’t a proper algorithm to find the optimal solution, though we are able to score the quality of a selected solution and decide whether it’s optimal.

Metaheuristics exploit a heuristic belief about the space of candidate solutions. This means that similar solutions behave similarly, thus small changes in parameters will result in small changes in the quality of the current solution as well. Class of well-known basic metaheuristics contains single-state methods (i.e. hill-climbing, simulated annealing or tabu search) and population methods (i.e.

genetic algorithms and evolution strategy from the field of evolutionary computation, or particle swarm optimization from the class of swarm intelligence methods). Furthermore countless combinations and modifications are described in the literature. In the current paper we won't go into details of these methods, an exhaustive and up to date description of metaheuristics is presented in [4].

Crucial stage of applying metaheuristics is the procedure of evaluating the candidate solution, which is usually executed by an objective function. Since this determines the difference between candidate solutions, it must reflects the quality of a candidate according to the given problem. Thus objective function is always problem-dependent.

Another question when applying metaheuristics is how to update the candidate solutions. In traditional problems the candidate solutions are represented as vectors, thus updating is executed by random alteration of values in the vector. Obviously, this is again problem-dependent, furthermore implementation-dependent. The idea we applied in case of metamodel-based test data generation is introduced in the following section.

B. Metaheuristics for test data generation

According to the test data generation idea outlined above, our task is to generate feasible program input using metamodel of test data, thus test data generation task can be interpreted as instance model generation.

The quality of generated test data is measured by an objective function. In our approach, we utilize the information given in form of specifications to construct the objective function, hence score the quality of a candidate solution. Thus, a candidate solution is considered better, if it covers more specification (if multiple specifications are considered in the objective function).

The specifications address the behavior of system under test, and not the test data directly, therefore they do not always carry information regarding input data (typically non-functional specifications define behavior independent from the input data). Obviously we must select, and even process suitable specifications to construct the objective function.

Since our candidate solutions are represented by instance models, updating of a candidate can be executed by model transformation. Possible transformations of candidate solutions are defined by a set of model transformation rules. In every iteration of test data generation, an arbitrary number of rules are selected and executed. Obviously, these rules don't violate the well-formedness rules provided by the metamodel. Figure 2 presents the workflow of the proposed test data generation algorithm.

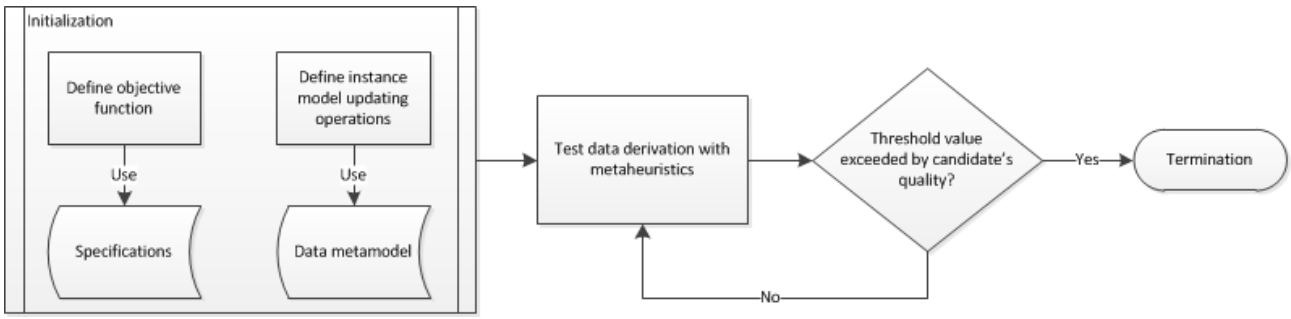

Figure 2: Workflow of test data generation algorithm.

IV. A simple example

In this section we show an example application of the proposed metamodel-based test data generation approach. Consider a software under test responsible for controlling an autonomous robot. This software is clearly an agent program, since it perceives it's environment through sensors and acts upon that environment through effectors (definition of agent in [5]).

Let's suppose that our system under test has an architecture presented in figure 3. The agent program maintains a model of its environment called the context model, and the agents uses this model as input data in order to decide among possible actions.

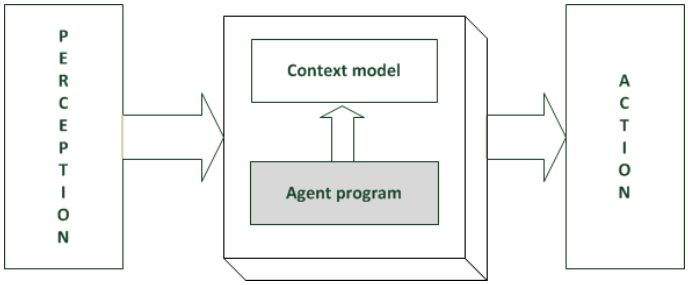

Figure 3: Architecture of the system under test.

Testing of the agent requires simulation of different environment settings, which can be performed by generation of context models, and injecting these test models into the architecture above in order to examine output of the agent program (feasible place of injection depends on the implementation).

Obviously this context model should have a metamodel, which describes the valid objects and relations within the environment of the agent. Our proposed test data generation algorithm could use this metamodel to generate complex test models.

Possible model transformations would include adding and erasing objects and relations from the metamodel, while an appropriate objective function could monitor, whether a necessary set of objects are present in the generated model (coverage criterion).

V. Conclusion and future work

In this paper we've introduced a novel metamodel-based test data generation method, which uses metaheuristic as test data derivation algorithm and constructs objective function from specifications.

In the outlined method, updating of candidate solutions represented by model instances is executed by model transformations. Unfortunately, constituting a set of transformation rules is labor intensive and time consuming. In the future, we plan to develop a method which automatically creates model transformation rules in run time, by partitioning the metamodel based on the information in specifications.

Beyond reducing the time needed to initialize a test data generation process, this solution would increase the universality of the approach, since this approach only requires formal specifications and formal metamodel to initiate test data generation.

References

[1] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Trans. Softw. Eng. Methodol.*, 5:63–86, January 1996.

[2] D. Neto, A. C., R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pp. 31–36, New York, NY, USA, 2007. ACM.

[3] J. Rushby, "Automated Test Generation and Verified Software," in *Verified Software: Theories, Tools, Experiments - First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, vol. 4171 of *Lecture Notes in Computer Science*, pp. 161–172. Springer-Verlag, 2008.

[4] S. Luke, *Essentials of Metaheuristics*, 2009, Available at http://cs.gmu.edu/∼sean/book/metaheuristics.

[5] S. Russell and P. Norvig, *Artifical Intelligence. A Modern Approach.*, Pearson Education Inc., second edition, 2003.