jPET: an Automatic Test-Case Generator for Java

Elvira Albert*, Israel Cabañas†, Antonio Flores-Montoya†, Miguel Gómez-Zamalloa*, Sergio Gutiérrez†
Complutense University of Madrid, Spain
*{elvira,mzamalloa}@fdi.ucm.es, †{antonioenriqueflores,sergio.gutierrez.mota,israelcabanasruiz}@estumail.ucm.es

***Abstract*—We present jPET, a whitebox test-case generator (TCG) which can be used during software development of Java applications within the Eclipse environment. jPET builds on top of PET, a TCG which automatically obtains test-cases from the *bytecode* associated to a Java program. jPET performs *reverse engineering* of the test-cases obtained at the bytecode level by PET in order to yield this information to the user at the source code level. This allows understanding the information gathered at the lower level and using it to test source Java programs.**

I. Introduction

Testing is considered nowadays a crucial discipline of software engineering. It has been proved to be one of the most expensive and time consuming ingredients in the software development cycle. This is mainly because of its enormous difficulty. Large sets of tests are needed to check the behavior of an application and, even so, bugs may remain undetected. This difficulty becomes more and more relevant as the software complexity grows and the security requirements increase. Therefore, there is a big need of developing tools that automate the task of generating unit tests that enable early and frequent testing while developing software.

A standard approach to generating test-cases statically is to perform a *symbolic execution* of the program [7], [3], where the contents of variables are expressions rather than concrete values. PET [5], [1] is test-case generator based on symbolic execution which receives as input a Java *bytecode* program and a selection of a coverage criterion. The coverage criteria are heuristics which try to estimate how well the program is exercised by a test suite. Examples of coverage criteria are *statement coverage*, which requires that each instruction of the code is exercised, *path coverage* which requires that every possible trace through a given part of the code is executed, and, *loop-k* which ensures a path coverage limiting the number of loop iterations to a threshold k. PET symbolically executes the bytecode and returns a set of *test-cases* associated to the branches of the execution tree built according to the coverage criteria. The test-cases are constraints over the input variables and the contents of the global memory (or *heap*) which contain the conditions to execute the different paths (or branches) of the tree. The constraints on the heap impose conditions on the shape and the contents of the heap-allocated data structures of the program. Concrete input values which satisfy the constraints are later obtained from the test-cases by using a constraint solver in order to actually produce unit tests. PET can be used from a basic command line or from a web interface, but none of them provide the adequate functionality to test Java applications during software development.

This paper presents jPET, an extension of PET which performs reverse engineering of the test-cases by taking the information gathered at the bytecode level and showing it in a comprehensible way at the Java source code level. For this purpose, jPET is integrated within the Eclipse programming environment and extends PET's functionality with the aim of helping developers test Java programs during software development. The main extensions, and contributions of this work, are summarized as follows:

- jPET incorporates a *test-case viewer* to visualize the information computed in the test-cases (includign the objects and arrays involved).
- It can display the test-case *trace*, i.e., the sequence of instructions that the test-case exercises.
- jPET can also parse method preconditions written in JML [6]. This can be very useful for avoiding the generation of uninteresting test-cases, as well as for focusing on (or finding) specific ones.

Also, these capabilities allow using jPET as a tool for performing *program comprehension* of both Java and Java bytecode programs at the method level. This can be done by observing (using the viewer and the traces explained above) the input-output behaviors of a method with a set of concrete executions. jPET is available for download as free software at the PET web site `http://costa.ls.fi.upm.es/pet/download`. Also, there is a demonstration video of the tool available at `http://costa.ls.fi.upm.es/pet/demo_video`.

II. Features of jPET and Implementation Details

The basic usage of jPET is as follows: once a Java source (or bytecode) file is opened, the user can select in the outline view of Eclipse the methods for which he/she wants to generate test-cases. Once selected, clicking on the jPET icon opens the preferences window of jPET which allows setting preferences such as the coverage criterion, whether to get concrete test-cases or just path constraints, the concrete numeric domain, etc. Currently, jPET provides two coverage criteria, *block-k* (an adaption of *loop-k* for bytecode [1]) and *depth-k* (which limits the number of executed instructions to a threshold). The obtained test-cases are then shown in a tree-like structure organized in packages, classes and methods in the jPET view.

A. The Test-Case Viewer

The output of PET was a textual (term-like) representation of the list of test-cases including the input and output heaps. Such representation quickly becomes unreadable, even

for rather simple heap shapes. The jPET's test-case viewer solves this issue by allowing the programmer to navigate through the input and output heaps graphically. The viewer is accessed by double-clicking on the selected test-case in the jPET view. The window basically has two main areas: One of them shows a table representation of the (input or output) heap and another table with the information of the currently selected object (including all its fields). The other area depicts a tree-like representation of the selected object and all transitively reachable objects from it. Each node in the tree can be expanded and contracted to show and hide its children. Furthermore, every node or edge can be moved to help the programmer interpret the results.

Implementation Details. The test-case viewer basically consists of three modules: (1) The output of PET has been extended so that an XML file containing all the information of the obtained test-cases is generated. (2) The `Graph Manager`, based on `JGraph` [2]manages the graphical representations based on the heap data structure. Each graphical representation is basically a graph that encapsulates the heap, or part of it, and contains information about how it should be displayed by the interface. (3) On top we have a `Swing` based interface that allows the programmer to interact with the `Graph Manager` by creating, displaying and exploring their preferred representations of the heap.

B. Displaying Test-Case Traces

jPET allows inspecting the trace of instructions that a given test-case exercises. This is done by right-clicking on a test-case and selecting "Show trace". The trace information can be displayed in two different ways: **(1)** jPET can highlight all lines (and hence instructions) that a test-case exercises. This is done by using the Eclipse highlighting features and markers which, resp., highlight the executed lines and report how many times, and in which order, these lines have been executed. This gives a global view of the test-case trace. We have taken advantage of the Eclipse environment that provides an API to color and mark code lines by using extension points such as "annotationTypes" and "markers". Therefore, we need to identify which Java lines are executed and send them to Eclipse. Since PET reasons over bytecode programs, the implementation of this feature has required to match bytecode instructions with lines in the source code. Also, we distinguish between normal execution and exception lines, showing them in different colors (green/red). **(2)** The second modality allows following the trace in the source code step by step by means of a debugger implemented using the debugging options of Eclipse. This functionality is started by right-clicking on a test-case and selecting "Show trace debug". The usual debugging actions can then be used to follow the trace.

C. Method Preconditions

One of the well-known problems of symbolic execution-based methods is the large amount of paths that have to be considered. This, on one hand, poses scalability problems mainly due to the memory requirements to build and maintain them. On the other hand, this complicates human reasoning on the obtained test-cases and, hence, their use within software testing during program development. One way to alleviate this situation is the use of *method preconditions* which allows the developer to specify conditions on the input arguments (and the corresponding reachable objects and arrays) which can prune the symbolic execution tree and therefore avoid the generation of certain useless test-cases, as well as focusing on (or finding) specific ones.

jPET allows writing preconditions using (a subset of) *JML*, the Java Modeling Language [6], which has become the standard specification language within software verification of Java. Preconditions are parsed by jPET and transformed into the internal notation of PET (in CLP [5]). Most preconditions can be expressed natively as constraints in the underlying constraint domain, in which case the paths that violate them can be pruned as soon possible (hence do not obtaining the associated test-cases).

III. Conclusions and Related Work

There exist many test-case generation tools, most of which are devoted to generate unit tests that can be used later for regression testing. However, only a few of them can be fully integrated within the software development process as jPET does. Among those, the most related ones are: PEX [8], a white-box unit-test generator for the .NET platform which includes graphical support through an add-in for Visual Studio; and KeY [4], which among its many features allows generating unit-tests for Java programs as well as visualizing their associated paths during symbolic execution. We argue that the high level functionality provided by jPET, namely the heap visualizer and the trace viewer, plays a fundamental role for the practical use of testing tools during software development. Furthermore, the fact that the jPET symbolic execution engine works at the bytecode level, allows using it for Java bytecode programs for which the source is not available. This could be very useful from the point of view of reverse engineering and, in particular, allows program comprehension of Java bytecode programs by means of observing the behavior of a set of concrete and meaningful executions.

References

[1] E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *Proc. of. PEPM'10*. ACM Press, 2010.

[2] David Benson and Gaudenz Alder. JGraphX (JGraph 6) User Manual. http://www.jgraph.com/doc/mxgraph/index_javavis.html.

[3] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[4] C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *Proceedings of TAP*, volume 4454 of *LNCS*. Springer, 2007.

[5] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *TPLP, ICLP'10 Special Issue*, 2010.

[6] The Java Modelling Language homepage. http://www.cs.iastate.edu/~leavens/JML/.

[7] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[8] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.