Delft University of Technology
Software Engineering Research Group
Technical Report Series

An Approach for Search Based Testing of Null Pointer Exceptions

Daniele Romano, Massimiliano Di Penta and Giuliano Antoniol

Report TUD-SERG-2011-028

TUD-SERG-2011-028

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the International Conference on Software Testing, Verification and Validation, 2011, IEEE CS Press.

© copyright 2011, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

An Approach for Search Based Testing of Null Pointer Exceptions

Daniele Romano[1], Massimiliano Di Penta[2], Giuliano Antoniol[3]

[1] Delft University of Technology, The Netherlands
[2] RCOST, University of Sannio, Italy
[3] SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada

E-mail: daniele.romano@tudelft.nl, dipenta@unisannio.it, antoniol@ieee.org

***Abstract*—Uncaught exceptions, and in particular null pointer exceptions (NPEs), constitute a major cause of crashes for software systems. Although tools for the static identification of potential NPEs exist, there is need for proper approaches able to identify system execution scenarios causing NPEs.**

This paper proposes a search-based test data generation approach aimed at automatically identify NPEs. The approach consists of two steps: (i) an inter-procedural data and control flow analysis—relying on existing technology—that identifies paths between input parameters and potential NPEs, and (ii) a genetic algorithm that evolves a population of test data with the aim of covering such paths. The algorithm is able to deal with complex inputs containing arbitrary data structures.

The approach has been evaluated on to test class clusters from six Java open source systems, where NPE bugs have been artificially introduced. Results show that the approach is, indeed, able to identify the NPE bugs, and it outperforms random testing. Also, we show how the approach is able to identify real NPE bugs some of which are posted in the bug-tracking system of the Apache libraries.

Keywords-**Null pointer exceptions; Search-based testing.**

I. Introduction

Uncaught exceptions are, very often, the cause of crucial problems for the security, safety and reliability of software systems. Specifically, the presence of such exceptions can cause system crashes, unintended behavior or, in some case, possibly lead an intruder to gaining unauthorized access to the system. In many critical software applications such as medical, aerospace, financial or high dependable software, uncaught exceptions may cause severe threats to human beings, as well as data or economic losses. A noticeable example of uncaught exception is the Null Pointer Exception (NPE), raised by the unexpected de-referencing of pointer variables not yet properly initialized. van Hoff [1] pointed out that the introduction of NPE problems is quite a typical mistake for Java developers, while Dobolyi and Weimer [2] proposed an approach to remove such a kind of problem from Java programs by transforming them to introduce error-handling code.

In the past, several approaches have been developed to statically analyze exception-related code [3], [4], [5], [6], [7]. In particular, Nanda and Sinha [4] proposed a context sensitive, scalable, accurate interprocedural null dereference analysis for Java systems. The static analysis is based on a backward path-sensitive algorithm that, starting from a variable use, backward traverses the interprocedural control flow graph, identifying paths possibly propagating to the use node a null pointer value. While static analysis techniques are often very thorough in identifying code likely raising an exception, to be conservative, they almost always produce as output a superset of likely dangerous paths. For this reason, testing activities are needed (i) to identify whether there are execution conditions actually raising such exceptions, and (ii) to better understand the context in which the exception is raised and thus easily fix it.

In this paper we propose to leverage the Nanda and Sinha analysis coupled with metaheuristic-based search to generate test input data leading to NPE. In a nutshell, our approach starts from the reduced set of paths identified by Nanda and Sinha analysis, possibly containing false positives, and uses a genetic algorithm to generate test input data causing a NPE in the unit under test. The algorithm is able to handle complex input data types (*e.g.*, strings and data structures).

It is the authors opinion that, even though the identification of potential NPEs using a static analysis approach would be 100% accurate (*i.e.*, no false positives), developers will in any case benefit from approaches that produce input data and scenarios leading to a program failure caused by the NPE. In fact, such input data would at minimum ease the cognitive burden and program understanding effort required to identify the conditions causing the NPE, and also facilitate the bug removal.

Our main contributions can be summarized as follows:

- we propose a novel approach to leverage accurate interprocedural null-dereference analysis and genetic algorithms to generate test input data leading to NPEs;
- we report empirical evidence—across 27 cases from 6 Java open source systems—of the capability of the proposed approach to detect artificially injected NPEs, as well as real NPEs documented in bug tracking systems. Also, we show that the approach is able to identify NPEs where a random testing fails, or at

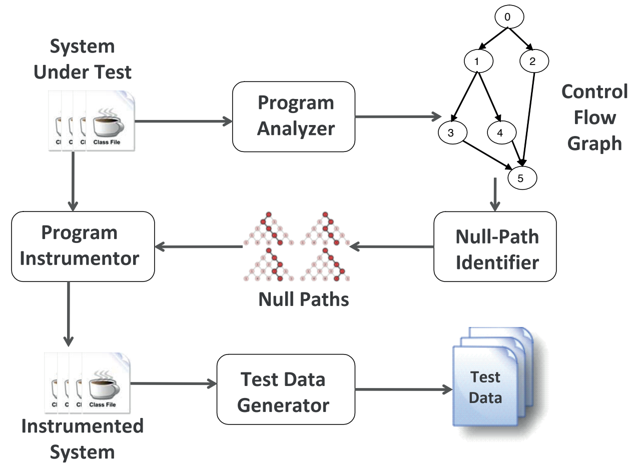


Figure 1. Overview of the proposed testing approach.

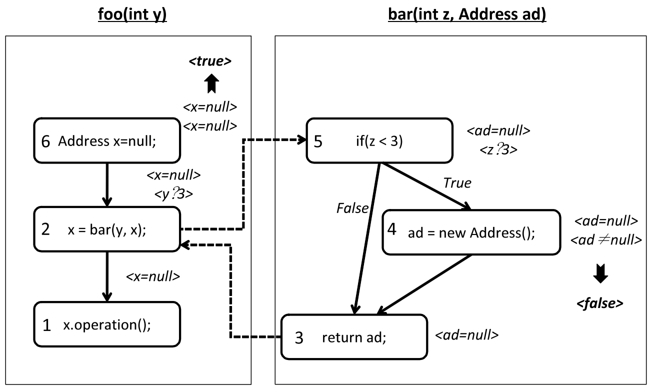


Figure 2. Example of null-path detection in a simple interprocedural control flow graph.

least requires to produce a significantly higher number of solutions (one order of magnitude higher, or even more).

The remainder of this paper is organized as follows. Section II describes the various steps of the proposed approach. The empirical study aimed at evaluating the approach performances is defined in Section III, while results are reported in Section IV, and Section V discusses the study threats to validity. After a discussion of the related literature (Section VI), Section VII concludes the paper and outlines directions for future work.

II. Approach

The test data generator we propose consists of four building blocks, shown in Figure 1: a program analyzer, a path selector, a program instrumentor and a test data generator. Although most of the testing approach is automated, its startup requires a manual intervention. This step consists in defining properly the boundaries of the System Under Test (SUT)—*i.e.*, in identifying the class cluster to be tested—and in identifying the the variable to be analyzed. Our testing process tests one variable at a time, with the aim of identifying NPEs that this variable can cause.

The first step of the process consists in building the interprocedural Control Flow Graph (CFG) of the SUT. This is accomplished by a program analyzer that receives as input the set of class files and generates as output the interprocedural CFG. Once obtained the CFG, the path selector identifies the set of *null-paths*, *i.e.*, paths that can dereference a null value and that can therefore cause a NPE. This is done by relying on an approach previously proposed by Nanda and Sinha [4]. After null paths have been identified, the objective of the test data generation process is the following: given a program P and a null-path u, generate a set of input parameter values x, such that u is traversed. Test data generation is performed using a Genetic Algorithm (GA), inspired by the structural coverage approach proposed by Baresel *et al.* [8] and also used by by Tracey *et al.* [9], [10]. Before executing the GA, the program is properly instrumented to allow the GA observing how far the inputs are from causing the execution of the target statement.

A. Interprocedural control and data flow analysis

The first step consists in constructing the interprocedural control flow graph of the SUT. This goal is achieved using the T.J. Watson Libraries for Analysis (WALA)[1]. WALA produces a *supergraph* of the SUT, which is a directed graph that connects CFGs of callers and callees. The *supergraph* is modified avoiding to expand exception related edges. This is to say, the interprocedural CFG does not expand the CFG of methods invoked in the *catch* statements.

B. Identification of null paths

This step aims at identifying all paths in the CFG that may dereference the variable of interest with a null value. To this aim, we used an approach previously proposed by Nanda and Sinha [4], which defined an algorithm for interprocedural null-dereference analysis for Java systems. Their approach consists of backward path-sensitive analysis, that starting at a dereference statement, where the variable of interest is dereferenced, propagates a set of abstract state predicates backward in the CFG.

Let us consider the example in Figure 2. The tester could be interested in analyzing the variable x used at node 1. The analysis starts at node 1 with the sole predicate<*x=null*>. When entering the procedure *bar* at node 2 of *foo*, a mapping between formal and actual parameters is performed, transforming the predicate in <*ad=null*>. When traversing node 4, the analysis detects an inconsistency among the predicates since the predicate <*ad≠null*> is added to the

[1]http://wala.sourceforge.net

set. Because of that, the path is marked as non-null-path and thus discarded. Instead, when traversing the node 5 through the false branch, the algorithm adds the predicate *<z≥3>* to the set. When the analysis returns to the procedure *foo*, it updates the predicates in *<x=null>* and *<y≥3>*. The analysis marks the path as null-path when node 6 is traversed, since the predicate *<x=null>* is added one more time to the set of abstract state predicates.

Even though this approach is accurate and reduces the number of false positives, it presents some approximations at program points where the source code is not available, *e.g.*, calls to external libraries. At such points the analysis applies the null-check rule that states: a reference *r* is assumed potentially *null* if it receives its value from outside the system boundary, and is checked against a *null* value.

```
public void foo(){
   String name=Library.bar();
   //code using name here
      …
}
```

(a) Without null-check rule

```
public void foo(){
   String name=Library.bar();
   if(name!=null){
      //code using name here
   }
   …
}
```

(b) With null-check rule

Figure 3. Detecting null paths when invoking libraries using null check rules.

Let us consider the method *foo()* shown in Figure 3-a. If the method *Library.bar()* returns a *null* value, its dereferencing will cause a NPE. To detect it, we transform the *foo()* code as shown in Figure 3-a, *i.e.*, applying a null-check rule—after the library invocation—to the variable *name*, which contains the return value of the library method *bar()*.

The use of this null-check rules could introduce a significant limitation in the detection of null-paths. To overcome such a limitation, we propose to transform the problem of letting a method invocation causing a NPE in a coverage problem. If we can derive from the library documentation/specifications the input conditions under which the method returns null value, the method invocation can be transformed into the following code:

```
Object x;
if(y.equals(criticalInputs))
  x=null;
else;
  x=foo(y);
```

where the method *foo* is the method under test, and *criticalInputs* are the inputs for which *foo* returns a null value.

C. Test data generation

The goal of the test data generator is to find input values that execute a null-path under test, and hence raise the NPE. The search process performed by the generator is accomplished through the execution of a GA. GAs are among several kinds of optimization methods, including Simulated Annealing, Tabu search, and several variations of them. Since the time GAs was proposed by Holland in the early 1970s [11], these algorithms have been used in a wide range of applications where optimization is required. The key idea of GAs is the analogy between the encoding of candidate solutions as a sequence of simple components, and the genetic structure of a chromosome.

Differently from other heuristics, instead of considering one solution at a time, a GA starts with a set of solutions, often referred to as individuals of a population. Since the search is based upon many starting points, the likelihood to sample more of the search space is higher than local searches. Solutions from one population are taken and used to form new populations, also known as generations. This is achieved using the evolutionary operators, and the process is repeated until some condition (*e.g.*, achievement of the goal) is satisfied. A GA is defined in terms of: (i) the encoding of candidate solutions, (iii) the definition of the fitness function, and (ii) the definition of the evolutionary operators.

Our GA implementation relies on the freely available GA framework *JMetal*[2]. In our approach the GA candidate solutions are inputs for the software under test (SUT). While most of the previous approaches for search based test data generation dealt with simple inputs, most of the existing programs have inputs consisting in complex data types. To deal with these inputs we decided to encode the data types with an XSD (XML Schema Definition) schema similarly to what previously done by Di Penta *et al.* [12] in an approach for service testing, where the inputs were encoded using the XSD contained in the service WSDL (Web Service Description Language) interface. Therefore, the solutions produced by the GA will be instances of XML files—represented as trees as shown in Figure 4—conform to an XSD document.

The second step defining the GA is the choice of the fitness function and of the selection operator, which determines the selection of the individuals to be reproduced.

The fitness function measures how close a solution is—*i.e.*, a set of inputs—to traverse the desired null-path and thus to reach the target statement and raise an exception. Our fitness function is inspired from the one adopted by Tracey *et al.* [9], [10] based on two distinct contributions: the

[2]http://jmetal.sourceforge.net/

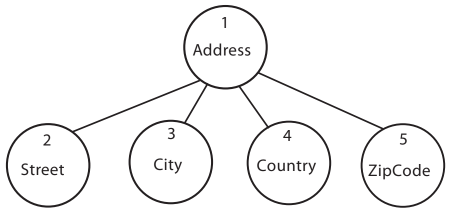


Figure 4. Tree-based genome representing complex inputs.

approaching level and the *branch distance*. The approaching level counts the number of critical nodes—*i.e.*, of predicate nodes that branches in the null path—successfully traversed by the solution, while the branch distance is the distance to satisfy a predicate in a critical node where an undesired branch is taken.

While Tracey *et al.* aggregated the two contributions in a unique fitness function, by weighting and summing them, we preferred to keep them separate and thus defined a comparison operator, for pairs (a_i, b_i), where a_i is the approaching level for solution i and b_i its branch distance. The comparison operator is defined as follows:

$$(a_1,b_1) > (a_2,b_2) = \begin{cases} true & if\, a_1 > a_2 \; or \\ & (a_1 = a_2 \; and \; b_1 < b_2) \\ false & otherwise \end{cases}$$

In other words, a fitness function pair is fitter than a second one if it has a bigger approaching level or, for an equal approaching level, it has a smaller branch distance.

Table I
BRANCH DISTANCES FOR PREDICATES AS DEFINED BY KOREL [13].

Predicate	Value
a>b	b-a-1
a≥b	b-a
a<b	a-b+1
a≤b	a-b
a=b	abs(a-b)

The branch distance computation strongly depends on the predicate to be satisfied. If the predicate is between numeric values we use the objective function for relational predicate defined by Korel [13], illustrated in Table I. If the predicate is expressed in terms of string comparison, the branch distance is computed following the character distance defined by Alshraideh and Bottaci [14]. They define the character distance as the sum of the absolute differences between the ordinal character values—in their representation, *e.g.*, ASCII—of corresponding character pairs. More precisely, given a pair of strings *s* and *t*, the Character Distance (CD) is defined as follows:

$$CD\,(s,t) = \sum_{i=0}^{i=k-1} |s_i - t_i| + 128 \cdot (l - k)$$

where x_i is the $i-th$ character of the string x, while l and k are the length of the two strings.

The selection operator selects two parent chromosomes from a population according to their fitness. We use a modified roulette wheel selection algorithm as proposed by Al Jadaan *et al.* [15], named Ranked Based Roulette Wheel (RBRW). It assigns to each individual a fitness value equal to its rank in the population: the highest rank has the highest likelihood to be selected. The probability a solution has to be selected is then computed as follows:

$$P_i = \frac{2 \cdot Rank}{N\_Pop \cdot (N\_Pop + 1)}$$

where N_Pop is the population size and $Rank$ is the rank of the individual in the population according to its fitness function.

Once the GA has selected the parents needed for generating the offspring, the crossover operator is applied to them with a probability P_c. We defined two different kinds of crossover operators:

1) a *leave crossover*, which selects leaves at the same level in two parents, applies one-point crossover to the leaves (which can be strings, integer or real numbers) and produces the offspring. One-point crossover between two strings s_1, s_2 is performed by cutting both strings at a random position $p < min\,(length(s_1), length(s_2))$ and exchanging the sub-strings after the cut-point. One-point crossover for numeric values is performed over their binary representations, using the default crossover operators defined in *JMetal*.
2) a *tree crossover*, which randomly exchanges sub-trees in two parents to produce the offspring.

After obtaining the offspring population though the crossover operator, the offspring is mutated through the mutation operator in order to ensure genetic diversity from one generation to the next ones. In particular, each individual is mutated with a probability P_m. For each individual to be selected for mutation, nodes can be mutated following three different strategies:

1) *AllNodes*, *i.e.*, all nodes are mutated;
2) *Random[1, nodes-ApproachLevel]*, the mutation is applied to a random set of nodes chosen in the interval $[1, nodes - ApproachLevel]$; where $nodes$ is the total number of nodes and $ApproachLevel$ is the approaching level currently reached. The rationale is that, when the approaching level increases, the mutation probability decreases, thus decreasing the probability of mutating inputs that help to branch in critical nodes of the null-path.

Table II
MUTATION OPERATORS.

Operator	Data Type	Description
Insertion	String	Inserts a random character into a given string. The insertion position is randomly chosen.
Deletion	String	Deletes a random character from a given string.
N-Substitution	String	Randomly replaces N contiguous characters in a string.
Swap	String	Randomly swaps two characters in a string.
Scramble	String	Scramble the position of the characters belonging to the whole string, or to one of its substrings.
Inversion	String	Reverses the order of characters between two random positions in the string.
Replace Contiguous	String	Replaces a randomly chosen character with a contiguous one in the ASCII table.
Jump	Numeric	Replaces a randomly selected digit with a new random digit.
Creep	Numeric	Adding a random (positive or negative) value.
Boundary	Numeric	Replaces the numeric value with either the upper or lower bound of the numeric field.

3) *1Node*, *i.e.*, only one randomly selected node is mutated.

Once a node is selected, the mutation is performed by randomly selecting a mutation operator—among those defined for the node type—from a pool, as shown in Table II.

D. Program instrumentation

As explained above, the program instrumentation block is needed to properly compute the fitness function value. This step is achieved instrumenting—in the Java byte code—all predicates belonging to the selected null-path. The instrumentor is based on the open source library Shrike part of the WALA framework.

III. EMPIRICAL STUDY

The *goal* of this empirical study is to evaluate the effectiveness of our approach against real software systems with complex input data. The *quality focus* are the ability of our approach to detect NPEs in real systems, and the ability of the developed metaheuristic search to find input data adequate to arise the NPEs. The *perspective* is that of researchers, interested in the integration of a static code analyzer with a search module able to find input data relevant to manifest the bugs identified by the analyzer. The results of our study could be of interest also for test and quality engineers who want to fortify the system under test removing, or at least narrowing, the likelihood of NPEs occurrence after the deployment of the SUT. The *context* of this study consists in six open-source systems, widely used in both academic and industrial community, that are: *DNSJava*, *Gnu Crypto*, *Apache TagLibs*, *Apache Batik*, *Apache Ant* and *Apache CommonsIO*.

Table III lists the class clusters under test derived from the six analyzed systems, where *className* is the class used as driver of the analysis, and *Method signature* is the signature of the method under test. The column *Artif. mutant* indicates if the code has been artificially mutated in order to generate the exception. Mutation was performed using a combination of the Kim *et al*. mutants for Java [16], with the aim of artificially introducing null pointer exception problems. In the other cases—*i.e.*, where no mutation was performed—we tested the program with the aim of finding documented NPEs already present in the source code.

The research question this paper aims at addressing is the following:

To what extent is the proposed approach capable of identifying NPEs in the programs under test?

To address such a research question, we analyzed the capability of the proposed approach to identify NPEs in the 27 analyzed cases, within a fixed, maximum number of GA fitness evaluations. Also, we compare the proposed approach with a random testing one, which uses the same mutation operator used by the GA to randomly generate test data (without however using the fitness function to guide the evolution). For cases where both GA and random testing converge, we compared the results of multiple runs of both algorithms by using a non-parametric test, namely the Mann-Whitney (unpaired) test.

Besides testing the presence of a significant difference between the two techniques, we also evaluated the magnitude of the difference using the Cohen d effect size [17]. For independent samples (as in our case) the effect size is defined as the difference between the means (M_1 and M_2), divided by the pooled standard deviation ($\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$) of both groups: $d = (M_1 - M_2)/\sigma$. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$ and large for $d \geq 0.8$ [17]. We chose the Cohen d effect size as it is appropriate for our variables (in ratio scale) and given the different levels (small, medium, large) defined for it, it is quite easy to be interpreted.

Besides addressing the research question from a quantitative point-of-view, as above described, we also report—in Section IV-A—a complete example, related to finding a real NPE bug occurred in the *Apache CommonsIO* library (SUT #27 in Table III). The aim of such example is to explain, step by step, how a tester can set-up a testing activity with the aim of finding such a kind of bugs.

A. Settings

We calibrated the GA and its operators as follows:

- the GA population is composed by 100 individuals, or candidate solutions. The initial population is randomly generated using the mutation operator;

Table III
DETAILS OF THE STUDIES CARRIED OUT.

ID	System	Class name	Method signature	Artif. mutant
1	DnsJava	Message	sectionToString(int)	Yes
2	DnsJava	Address	parseV4(String)	No
3	DnsJava	ExtendedResolver	getResolver(int)	No
4	DnsJava	Record	getbyteArrayFromString(String s)	Yes
5	DnsJava	Name	fromDNAME(DNAMERecord)	Yes
6	DnsJava	CertRecord	string(int)	Yes
7	GNU Crypto	KeyPairCodecFactory	getInstance(int[5] buffer)	No
8	GNU Crypto	KeyPairCodecFactory	getInstance(String)	Yes
9	GNU Crypto	KeyPairCodecFactory	getInstance(Key)	Yes
10	GNU Crypto	KeyPairCodecFactory	getInstance(Key)	Yes
11	GNU Crypto	KeyAgreementFactory	getPartyAInstance(String)	Yes
12	GNU Crypto	KeyPairAgreementFactory	getPartyBInstance(String)	Yes
13	GNU Crypto	KeyPairCodecFactory	getInstance(Key)	Yes
14	GNU Crypto	KeyPairGeneratorFactory	getInstance(String)	Yes
15	GNU Crypto	Properties	remove(String)	No
16	GNU Crypto	SeverMechanism	getNegotiatedProperty(String)	Yes
17	Apache TagLibs	Coercions	coerceToObject(String)	Yes
18	Apache TagLibs	Util	getScope(String)	Yes
19	Apache TagLibs	Util	getContentTypeAttribute(String,String)	Yes
20	Apache TagLibs	Util	getStyle(String,String)	Yes
21	Apache Batik	CSSUtilities	convertPointerEvents(int)	Yes
22	Apache Batik	CSSUtilities	convertShapeRendering(int)	Yes
23	Apache Ant	JspC	execute(String)	Yes
24	Apache Ant	Javac	getAltCompilerName(String)	Yes
25	Apache Ant	Javac	getAltCompilerName(String)	Yes
26	Apache Ant	MimeMailer	parseCharSetFromMimeType(String type)	Yes
27	Apache CommonsIO	FilenameUtils	equalsNormalizedOnSystem(String,String)	No

- the crossover probability is 0.9, while the mutation probability is 0.9/(size of the individual representation). Such a size depends on the tree structure and leave representation. These were default values for *JMetal*, except for the default mutation probability that was 1/(size of the individual representation).
- for string mutation the length of the randomly generated strings is randomly chosen in the range [0,30];
- for numeric mutation, the number of digits is randomly chosen in the range [1,5];
- the maximum number of fitness evaluations before the GA (or the random testing) terminates without finding a NPE is 10,000. We actually tried for many of the programs up to 100,000 evaluations, which however did not produce better result in terms of random test success nor of better results for GA.

Specifically, GA population size, mutation and crossover probability were chosen using a trial-and-error procedure aimed at selecting values that, for our case studies, allowed to obtain the best results. Mutation settings were chosen to be acceptable for the inputs of the tested programs. Finally, the maximum number of evaluations was chosen high enough to avoid a ceiling effect, *i.e.*, having a too low value that could have led to results strongly dependent on the convergence capability of the approach. Finally, to narrow the natural randomness of GAs, every experimentation has been repeated ten times, and the results of the different instances have been analyzed through proper statistical procedures, as described above.

B. Choosing the proper crossover and mutation operators

To evaluate the effectiveness of the different mutation and crossover operators, we used a toy program, described in the appendix of a longer technical report[3]. The rationale of using an artificial program is to have an example with a high number of parameters (*i.e.*, input arguments) entailing a large and complex search space. In fact, the program requires as input a data structure of nine elements, including strings and integers. To generate the NPE, a combination of seven conditions imposed on six of the input parameters must be met, meaning that, to reach the NPE, an approaching level of seven must be reached.

From the results of the small calibration studies (see the technical report) concerning the mutation operators we found out:

1) The *AllNodes* operator applies the mutation to all nodes in the tree and, despite being faster to satisfy the first predicate it encounters, it can block the search when attempting to satisfy the subsequent predicates. This is because if, every time, we mutate again all nodes, this does not guarantee that previously satisfied predicates will still be satisfied.
2) The *Random(1, nodes-approachLevel)* applies the mutation to a number of nodes randomly chosen in the interval *[1, nodes-ApproachLevel]*, where *nodes* is the number of tree nodes for the individual being mutated, and *ApproachLevel* is the number of satisfied predicate

[3]http://web.soccerlab.polymtl.ca/ser-repos/public/NPE-testing-tr.pdf

nodes in the null-path being covered. Differently from the previous mutation operator, this operator does not block the search, and is therefore able to reach the last predicate. However, the number of evaluations required increases when increasing the approaching level.

3) Finally, the *1node* operator exhibit the best performances, as (i) it is able to reach the last predicate and (ii) the number of required evaluations does not increase with the approaching level.

For what concerns the crossover operators, we found that the *tree crossover* required a significantly lower number of evaluations than the *leave crossover*, as it operates on the entire tree instead of on a single leaf, thus producing a higher genetic diversity than the leaves crossover.

In conclusion, the small calibration study suggests us to use, for our empirical study, the *1node* mutation operator and the *tree* crossover operator.

IV. Results

This section reports results of our empirical study. The study raw data are available for replication purposes[4].

Table IV reports the number of evaluations (averaged on the 10 runs performed) required by GA and random search respectively, to find a NPE in the 27 analyzed cases. As the table shows, in 21 out of 27 cases random search was not able to find any NPEs within the maximum number of evaluations we considered. In 6 cases random search was able to find NPEs, however (i) only for some of the runs (percentage indicated in parentheses, and ranging between 10% and 40% of the runs), and (ii) the number of evaluations required by random search was about one order of magnitude—or in some cases even more—higher than GA. It is worth noting that random search suceeded only when the input parameters are integer, while failed for complex inputs, thus highlighting that, in many real cases—where inputs parameters are strings or complex data structures, random testing does not represent a viable solution.

Figure 5 shows boxplots comparing—for the cases where random search succeeded—the distributions of number of evaluations for both GA and random testing, and Table V reports the results of the Mann-Whitney tests, as well as the Cohen d effect size. The table shows that:

1) the p-values are significant for SUT #3, #6, #7, #21, while they are not significant for SUT #1 and #22. However, it should be noticed that the lack of significance in these cases mainly depends on the limited number of cases for which random testing converged;
2) the effect sizes are high (> 1 for SUT #3, #6, #7, and #21), and medium for SUT #1. Overall, both effect sizes and the average number of evaluations shown in Table IV clearly indicate the superiority of the GA-based approach.

[4]http://web.soccerlab.polymtl.ca/ser-repos/public/NPE-testing-rawData.tgz

Table IV
Number of evaluations required on average by GA and random search to find a NPE.

SUT ID	System	GA	Random
1	DNSJava	198	853(40%)
2	DNSJava	422	–
3	DNSJava	206	1953 (20%)
4	DNSJava	458	–
5	DNSJava	562	–
6	DNSJava	384	3765 (30%)
7	GNU Crypto	216	2404 (30%)
8	GNU Crypto	408	–
9	GNU Crypto	609	–
10	GNU Crypto	880	–
11	GNU Crypto	374	–
12	GNU Crypto	564	–
13	GNU Crypto	384	–
14	GNU Crypto	432	–
15	GNU Crypto	571	–
16	GNU Crypto	6510	–
17	Apache TagLibs	240	–
18	Apache TagLibs	883	–
19	Apache TagLibs	4506	–
20	Apache TagLibs	1267	–
21	Apache Batik	286	4207 (20%)
22	Apache Batik	246	6114 (10%)
23	Ant	373	–
24	Ant	1299	–
25	Ant	2726	–
26	Ant	1076	–
27	CommonsIO	1940	–

Table V
GA vs random testing: results of Mann-Whitney test and Cohen d effect size.

SUT ID	M-W p-value	Cohen d
1	0.45	0.72
3	0.030	2.01
6	0.0060	1.36
7	0.0060	1.66
21	0.040	1.78
22	0.13	–

A. A complete example: finding an Apache CommonIO NPE bug

This section describes, step by step, how the proposed approach can be used to find a NPE bug using the proposed approach. Let us consider the bug of *Apache CommonsIO (1.2, 1.3, 1.3.1, 1.3.2)* posted in the Apache Jira server with id IO-128 (our SUT #27). According to the bug description, the execution of the following statement:

```
FilenameUtils.equalsNormalizedOnSystem(
                String a, String b);
```

could throw a NPE at line 970 when the following statement—contained in in the *equals* method—is executed:

```
filename1.equalsIgnoreCase(filename2);
```

Manually finding input values raising this exception could be quite difficult. In fact, this remained in the *Apache Common-*

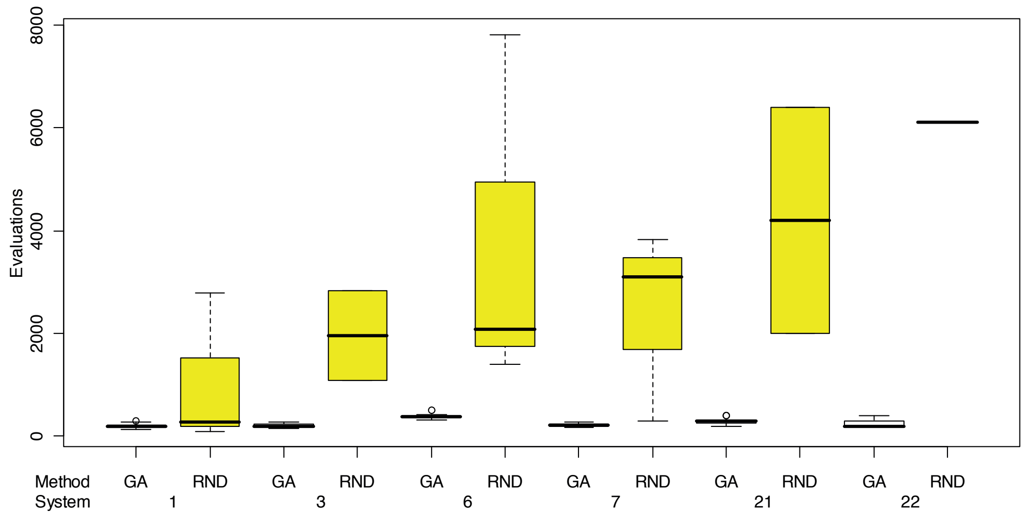


Figure 5. Comparison between GA and random testing: boxplots.

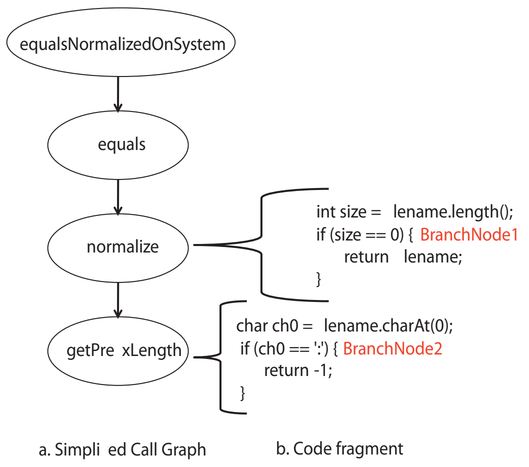
sIO library for four releases without being discovered. Also, it was reopened several times in the Apache bug tracking system.

In order to apply our approach to find the input value necessary to throw that NPE, the first step is to properly mark the variable of interest, in this case the variable *filename1*:

```
@check(filename1)
filename1.equalsIgnoreCase(filename2);
```

To understand how the NPE can be raised, let us consider a simplified call graph with the related code fragments as shown in Figure 6. The *null path identifier* detects the null-path that traverses the *false* branch in the predicate node 1, and the *true* branch in the predicate node 2. Let us suppose such a null-path be composed by only these two critical nodes (by ignoring all other statements that do not have an influence on these predicates). We instrument the conditionals to compute the fitness value for each candidate solution of the GA, and then the GA can start. With the aim of explaining how the fitness function works to guide the GA towards finding the inputs raising the NPE, Table VI shows the fitness values for some candidate solutions, where *a_level* is the *approaching level* and *b_distance* is the *branch distance*. As shown, for an empty input string the *approaching level* is still 1, while it increases to 2 for a non-empty string. Then, as the ASCII code of the character decreases—from lowercase to uppercase letters, until reaching special characters and thus the colon “:”—the branch distance decreases accordingly.

For this case the algorithm took, on average, 1,940 fitness evaluations and 329 seconds to find the NPE (timing perfor-



```
int size =  lename.length();
if (size == 0) { BranchNode1
    return  lename;
}
```

```
char ch0 =  lename.charAt(0);
 if (ch0 == ':') { BranchNode2
    return -1;
}
```

Figure 6. BUG IO-128 in Apache Commons IO: simplified call graph of the code under test.

mances have been evaluated on a MacBook 2,1, processor 2.0 GHz Intel Core 2 Duo, memory 2 GB 667 MHz DDR2 SDRAM, OS X leopard).

V. Threats to Validity

This section discusses the threats to validity that can affect the empirical study described in the previous sections.

Threats to *construct validity* concern the relationship between theory and observation. In our study this threat can be due to the fact that most of the bugs were artificially

Table VI
FITNESS VALUES FOR SAMPLE CANDIDATE SOLUTIONS FOR SIMPLIFIED VERSION OF IO-128.

Individual	*filename* variable value	Fitness (approaching level, branch distance)
1	””	(1,1)
2	”a”	(2,39)
3	”M”	(2,19)
4	”A”	(2,7)
5	”;”	(2,1)
6	”:”	(2,0)

introduced in the system, and thus they might not properly represent the reality. However, bugs were introduced using combinations of well-known mutation operators for Java defined by Kim *et al*. [16], that have been previously used to evaluate the effectiveness of testing strategies for object-oriented programs [18].

Internal validity threats concern external factors that may affect an independent variable. We limited the bias of GA (and random search) intrinsic randomness of our results by repeating each experiment 10 times and using proper statistics to compare the results. We have calibrated the GA settings using a trial-and-error procedure, and chosen the crossover and mutation operators by doing a small study on a toy program: although we found evidence about the superiority of specific operators and were able to understand the reasons of that, it can happen that (i) studies on different programs could lead to a different choice of the crossover and mutation operators, and (ii) the obtained calibration may not be the most suitable for our subject programs.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. Wherever possible, we used proper statistical tests to support our conclusions for the two research questions, in particular non-parametric tests which do not make any assumption on the underlying data distribution. It is important to note that, although we perform multiple Mann-Whitney tests, p-value adjustment (*e.g.*, Bonferroni) is not needed as we are performing the test on independent, disjoint data sets. Finally, we also reported the practical significance of the differences using the Cohen d effect size.

Threats to *external validity* concerns the generalization of our findings. We evaluated the proposed approach over 27 cases coming from 6 different Java systems. Nevertheless, a larger evaluation would be highly desirable.

VI. RELATED WORK

Our approach relies on the work of Nanda and Sinha [4], that proposed an inter-procedural null dereference analysis for Java systems, implemented in a tool called XYLEM. Their work consists in an interprocedural, context-sensitive and path-sensitive backward analysis, that starting from dereferencing statements identifies true and false null-propagation paths. The empirical studies conducted by Nanda and Sinha demonstrated the effectiveness of their approach in detecting bugs that many static analyzer tools miss, at the same time reducing the false positive rate. While we share with Nanda and Sinha the idea of identifying potential null dereferencing paths through interprocedural static analysis, our work is complementary to their one as we aim at identify inputs that actually cover such null paths. In fact, there may be cases where the null path is unfeasible or protected by proper pre-conditions, thus the result of static analysis may just reveal a warning of no particular interest.

In recent years, several researchers have analyzed the effect of exception handling constructs in Java programs. Ryder *et al*. [19] proposed a tool named JESP, that is aimed at examining the usage of user thrown exceptions in Java source code. Sinha and Harrold [6] presented techniques to construct representations for programs in presence of exception handling constructs, focusing on the effects of exceptions on program-analysis techniques, such as control flow analysis, data flow analysis, and control dependence analysis. Robillard and Murphy [5] and Jo *et al*. [3] conducted studies analyzing exception flows in Java. Specifically, Robillard and Murphy developed a tool named Jex exception flow and to generate a view of the exception structure, while Jo *et al*. proposed a static analysis approach taking into account the uncaught exceptions independently of the programmer's declarations. Sinha *et al*. [7] proposed an integrated approach for providing automated support for the testing of programs that contain implicit control flow caused by exception handling and polymorphism. All the above mentioned works propose static analysis approaches, while our work combines static analysis, previously proposed by Nanda and Sinha [4] with a search-based testing approach.

Several researchers have been focusing on the techniques for testing exceptions and recovery code. Fu *et al*. [20] focused on the robustness testing of Java server-side applications. They proposed a white-box coverage testing of exception handlers, that consists in the instrumentation of code at compile-time to inject exceptions and in the recording the handlers exercised.

The closest work to ours is the one of Tracey *et al*. [9], [10], who developed an automated test-data generator to test exceptions handling code using Simulated Annealing and Genetic Algorithms. While there are many similarity between the the approach of Tracey *et al*. [9], [10] and ours, there are also noticeable differences:

- our approach integrate the static interprocedural analysis of Nanda and Sinha [4] with the testing approach, with the aim of identifying the paths that could throw a NPE. This allows for easily setting the testing target, *i.e.*, the path to be traversed.
- while the work of Tracey *et al*. deals with testing exception handling code, we focus on generating input data that cause NPEs;

- our approach foresees the possibility of generating input data for programs having as input parameters complex data structures. This is achieved using a tree-based representation of inputs, as well as proper crossover and mutation operators for such a representation, and, finally, the operators suggested by Alshraideh and Bottaci [14] to handle branch distance in terms of string comparisons.

VII. Conclusion

This paper proposed a search-based approach aimed at identifying uncaught null-pointer exceptions (NPEs) in Java applications. The approach combines an interprocedural analysis previously proposed by Nanda and Sinha [4], aimed at identifying paths potentially dereferencing null pointers, with a Genetic Algorithm (GA) that generates test data with the aim of covering such paths. The GA allows for generating test data for program requiring as inputs complex data structures.

Results of the empirical study we performed on 27 cases from six open source Java applications showed the capability of the proposed approach to identify both artificially introduced and real NPE problems. Also, the empirical study showed that the approach significantly outperforms a random test data generation, which, in most cases, is not able to identify the problem within a large number of evaluations (10,000).

Future work aims at: (i) further validating the proposed approach on a larger set of cases, including more real-world bugs; (ii) better supporting cases where NPEs originate from invocations of library class methods, by specifying sets of pre and post conditions for such libraries; and (iii) extending the approach to support the testing of other kinds of exceptions, besides NPEs.

References

[1] A. van Hoff, "The case for Java as a programming language," *IEEE Internet Computing*, vol. 1, no. 1, pp. 51–56, 1997.

[2] K. Dobolyi and W. Weimer, "Changing Java's semantics for handling null pointer exceptions," in *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*. IEEE Computer Society, 2008, pp. 47–56.

[3] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe, "An uncaught exception analysis for java," *Journal of Systems and Software*, vol. 72, no. 1, pp. 59–69, 2004.

[4] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 133–143.

[5] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 191–221, 2003.

[6] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Trans. Software Eng.*, vol. 26, no. 9, pp. 849–871, 2000.

[7] S. Sinha, A. Orso, and M. J. Harrold, "Automated support for development, maintenance, and testing in the presence of implicit control flow," in *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, 2004, pp. 336–345.

[8] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information & Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[9] N. Tracey, J. Clark, K. Mander, and J. McDermid, "Automated test-data generation for exception conditions," *Softw. Pract. Exper.*, vol. 30, no. 1, pp. 61–79, 2000.

[10] N. Tracey, J. Clark, J. McDermid, and K. Mander, "A search-based automated test-data generation framework for safety-critical systems," vol. Systems engineering for business process change new directions, pp. 174–213, 2002.

[11] HollandJ.H., *Adaptation in Natural Artificial Systems*. University of Michigan Press, 1975.

[12] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno, "Search-based testing of service level agreements," in *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*. ACM, 2007, pp. 1090–1097.

[13] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[14] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Softw. Test. Verif. Reliab.*, vol. 16, no. 3, pp. 175–203, 2006.

[15] C. R. O. Al Jadaan, L. Rajamani, "Improved selection operator for ga," *Journal of Theoretical and Applied Information Technology*, vol. 4, no. 4, 2008.

[16] S. Kim, J. Clark, and J. McDermid, "The rigorous generation of java mutation using HAZOP," in *Proc. 12th Int'l Conf. Software and Systems Eng. and Their Applications (ICSSEA 99)*, 1999.

[17] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Lawrence Earlbaum Associates, 1988.

[18] L. C. Briand, M. Di Penta, and Y. Labiche, "Assessing and improving state-based class testing: A series of experiments," *IEEE Trans. Software Eng.*, vol. 30, no. 11, pp. 770–793, 2004.

[19] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JESP," in *CC*, 2000, pp. 67–81.

[20] C. Fu, A. Milanova, B. G. Ryder, and D. Wonnacott, "Robustness testing of Java server applications," *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 292–311, 2005.

TUD-SERG-2011-028
ISSN 1872-5392