Leveraging a Constraint Solver for Minimizing Test Suites

José Campos
Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
jose.carlos.campos@fe.up.pt

Rui Abreu
Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
rui@computer.org

***Abstract*—Software (regression) testing is performed to detect errors as early as possible and guarantee that changes did not affect the system negatively. As test suites tend to grow over time, (re-)executing the entire suite becomes prohibitive. We propose an approach, RZOLTAR, addressing this issue: it encodes the relation between a test case and its testing requirements (code statements in this paper) in a so-called coverage matrix; maps this matrix into a set of constraints; and computes a collection of optimal minimal sets (maintaining the same coverage as the original suite) by leveraging a fast constraint solver. We show that RZOLTAR efficiently (0.95 seconds on average) finds a collection of test suites that significantly reduce the size (64.88% on average) maintaining the same fault detection (as initial test suite), while the well-known greedy approach needs 11.23 seconds on average to find just one solution.**

***Keywords*-Regression testing, test suite reduction, constraint solver, fault detection, empirical evaluation.**

I. INTRODUCTION

In recent years, the software testing research community has given considerable attention to the subject of regression testing. Some approaches [1], [2], [3], [4], [5], [6], [7], just like the one proposed in this paper, targeted cost reduction of regression testing by selecting only a few test cases from the original test suite. Approaches for *test suite minimization* trade-off completeness (in terms of some criteria, such as code coverage) for time efficiency (e.g., [1], [2], [4]).

Each test case can be conceptually viewed as an artifact that will test a *set* of requirements (e.g., the test case executes/verifies components `foo` and `bar`). Mapping a test suite into a collection of sets reduces the problem of test suite minimization to the minimal hitting set problem [8]. Despite being an NP-hard problem [8], [9], recent advances in the AI community have proposed constraint solvers that are fast and scale to millions of variables (consequently, to large software programs) [10].

Our approach to minimize test suites, dubbed RZOLTAR, leverages the efficient and scalable off-the-shelf MINION constraint solver [10]. Our approach differs from related work because: (i) It does not trade-off completeness for time efficiency (e.g., greedy heuristic approach [1], [4]); (ii) it produces a collection of minimal[1] sets (and not only just one like the greedy or the integer programming approach [5]). As our approach yields more than one minimal set, the developer can then *prioritize* them using, e.g., cardinality or time to execute the test cases.

The adoption of regression testing techniques, such as minimization, remain limited, as there are only a few tools providing state-of-the-art techniques. To facilitate wide adoption of our technique, we implemented RZOLTAR within the GZOLTAR toolset [11]. GZOLTAR is an Eclipse[2] plug-in for automatic debugging, offering graphical visualizations of diagnostic reports produced by Spectrum-based fault localization (SFL) [12]. The main reason for this choice is to allow developers to adopt the tool without much effort as it is already integrated into the Eclipse IDE and takes as input JUnit[3] test cases.

We have evaluated the performance of our approach using several open source, real, and large software programs. Our empirical evaluation indicates that RZOLTAR can indeed significantly reduce the original test suite. We observed average reductions of 64.88% in terms of number of test cases and 45.59% of time reduction, while still maintaining full code coverage as the initial test suite.

Comparing with the well-known greedy approach (as described in [4]) which is still amongst the best performing heuristics, RZOLTAR reduces the size of the original suite more than greedy, and provides more than one minimal set for almost all open-source projects used in the evaluation in less time (11.9 times on average). RZOLTAR can also guarantee the same fault detection as the initial test suite (which means a *Reduction in Fault-Detection Capability* (RF)[4] value of 0.0%) for all subjects, whereas the greedy approach has a significant reduction of RF (88.16% on average). This paper makes the following contributions:

- We propose a technique for test suite minimization based on constraint solving programming, which efficiently reduces the size of the test suite, maintaining full coverage and fault detection rate;
- The proposed technique has been implemented within the GZOLTAR toolset [11], more specifically in a cutting-edge Eclipse view dubbed RZOLTAR, this way providing an ecosystem for testing and debugging software programs;

[1]Minimal in terms of the number of test cases needed to achieve the same code coverage.

[2]The Eclipse Foundation website, http://www.eclipse.org/, 2013.

[3]JUnit's official website, http://www.junit.org/, 2013.

[4]Formally defined in Section IV.

- We empirically evaluate the test minimization capabilities and fault detection of RZOLTAR using large, real world software programs;
- We compare the performance and results of our approach with greedy, known as being an effective time algorithm [4].

To the best of our knowledge, our approach to test suite minimization has not been described before.

II. PRELIMINARIES

In this section we introduce the relevant concepts for this paper and a motivational example, which is used throughout this paper.

Definition 1 (Program) *A program* Π *is a collection of* M *components,* $C = \{c_1, \ldots, c_j, \ldots, c_M\}$*, implementing a specific set of specifications and requirements.*

Consider the source code Π of a software program (see Fig. 1 for our running example, where we consider three components: line 1 is component c_1, line 2 is component c_2, and line 3 is component c_3). During the development phase of the software development life-cycle it is commonly to have a set of testing requirements for Π. Note that it is irrelevant for our approach what requirements are, but in the context of this paper we assume source code statements.

Definition 2 (Test Case) *A test case* t *is a* (i, o) *tuple, where* i *is a collection of input settings or variables for determining whether a software system behaves as expected or not, and* o *is the expected output. If* $\Pi(i) = o$ *the test case passes, otherwise fails.*

Definition 3 (Test Suite) *A test suite* $T = \{t_1, \ldots, t_i, \ldots, t_N\}$ *is a set of* N *test cases that are intended to test whether the program follows the requirements.*

As an example consider the following test suite $T = \{t_1, t_2, t_3, t_4\}$. Test case t_1 checks whether `add(1,2)` and `sub(2,1)` follow the specification or not, test case t_2 checks `add(1,0)` and `mul(1,0)`, test case t_3 checks `sub(1,0)`, and test case t_4 checks `mul(0,1)`.

During the development phase of software, a common practicing to build and maintain a *regression test suite* [13], [14]. Such test suite is used to perform regression testing of the software after a change is made (e.g., after fixing a bug or adding a new feature). Regression test suites are therefore an important artifact of the software development process to assess the quality of the software. Moreover, as developers often add new test cases to the suite as the development progresses, it must be maintained in order to keep testing efficiency optimized.

In many situations (e.g., testing that requires user input) the size of the test suite may be simply too large, making it impractical to execute all the test cases in the suite. Besides, the ever increasing time-to-market pressure requires the testing phase to be optimized as much as possible. Therefore, to make regression testing amenable to large programs, the suite should be minimized, i.e., it should contain only the necessary test cases to test the requirements and discard redundant ones. As an example, one can easily conclude that there is no need to execute all test cases in the previous test suite T as using, either t_1 **and** t_4 or t_2 **and** t_3 already checks all components.

```
   public class Calculator {
1.    public int add(int x, int y) { return x + y; }
2.    public int sub(int x, int y) { return x - y; }
3.    public int mul(int x, int y) { return x * y; }
   }
```

Figure 1: Example Program.

To minimize the number of tests but still achieve the same code coverage as the original suite, we consider that the testing requirements for each test case can be encoded as a set. For example, $\{c_1, c_2\}$ are the testing requirements for t_1. Taking as input a collection of sets, one per test case, the problem boils down to find the minimal set of tests cases that achieve the same coverage as the original test suite. In other words, the problem is to find the minimal hitting set of the collection of covered requirements set. A minimal hitting set can be defined as follows

Definition 4 (Minimal Hitting Set) *Let* S *be a collection of* N *non-empty sets* $S = \{s_1, \ldots, s_N\}$*. Each set* $s_i \in S$ *is a finite set of components (source code statements in our case), where each of the* M *components is represented by* c_j*. A minimal set coverage of* S *is a non-empty set* T' *such that*

$$\forall s_i \in S, (\ s_i \cap T' \neq \emptyset\ ) \wedge (\ \nexists T'' \subset T' : s_i \cap T'' = \emptyset\ ) \quad (1)$$

i.e., each member of S has at least one component of T' as a member, and no suitable subset of T', T'', is a hitting set.

Identifying the minimal hitting set (also know as dual set cover problem [8]) of a collection of sets is an important problem in many domains (e.g., such as Air Crew Scheduling [15]). During the construction of the minimal hitting set, each test should be analyzed to determine which components it covers. All t_i in T', must cover at least one component. Being an NP-hard problem (i.e., exponential on the number of components) [8], exhaustive algorithms (e.g., [16], [17], [18]) are prohibitive for real-world, often large programs. Furthermore, heuristic approaches (e.g., approach [1], [4]) trade-off completeness for time efficiency.

In the next section, we propose our approach to efficiently minimize test suites which guarantee to cover exactly the same requirements as the original suite. Moreover, our approach, unlike most related work approaches, generate multiple minimal sets for the minimization problem. As multiple minimal sets are generated, the developer can prioritize them using two criteria: (i) cardinality of the minimized test suite or (ii) test suite's execution time.

III. RZOLTAR

In this section we describe our approach, coined RZOLTAR, a constraint-based technique for test suite minimization. RZOLTAR takes as input the testing requirements

for each test case and yields a collection of minimized test suites that guarantee both the same coverage and the same fault detection as the initial test suite. The approach works in three major phases:

1) It executes the system under analysis with the current test suite in order to obtain the so-called coverage matrix (see Section III-A);
2) The coverage matrix is subsequently converted into a set of constraints, which are amenable to be solved by a constraint solver (detailed in Section III-B);
3) The constraints are solved with the slightly modified, off-the-shelf constraint solver MINION, and prioritized using a certain criterion (detailed in Section III-C).

A. Test Case Coverage

As mentioned before, although our approach is not limited to any testing requirement, in this paper we use code coverage. In the following we detail how code coverage is stored for subsequent usage[5].

By tracking which components each test case activate (covers) and the pass/fail result of that test, a $N \times M$ binary matrix A and an error vector e (with the information if the test execution has passed or not) is created. We refer to this matrix as *coverage matrix*. An element a_{ij} is equal to 1 if and only if component c_j is covered when the test t_i is executed

$$\omega(t_i,\ c_j) = (a_{ij} == 1\ ?\ true : false) \tag{2}$$

Note that, while a test may be designed to cover a specific component, other components may still be covered. Thus, the result of interception between test and component, returned by the $\omega(t_i,\ c_j)$ function in Eq. (2), can be explored to reduce the number of tests to generate a test suite that leads to the same coverage as the initial test suite. Next, we elaborate on how to exploit the information in the coverage matrix to minimize and prioritize the current test suite.

B. Modeling Coverage Matrix as Constraints

As mentioned before, our approach minimizes the current test suite using a constraint solver. In order for the coverage matrix to be amenable to off-the-shelf, fast, and scalable constraint solvers, RZOLTAR converts the coverage matrix (which contains all information available as no other modeling is required) into a set of constraints.

The key idea behind our approach is to encode the testing requirement of each test case into a set of constraints (see Algorithm 1), each of which has to be satisfied for the problem to be solved. For ease of comprehension, we illustrate this phase through an example. Consider the example in Fig. 2, which is obtained by running four test cases which cover the three components of the program in Fig. 1. The three components in Fig. 2 represents the three statements of the example program: c_1 correspond to statement with number `1`, c_2 to statement `2`, and c_3 to statement `3`. The four test cases execute at least one component: t_1 exercises components c_1 and c_2, t_2 covers components c_1 and c_3, t_3 exercises component c_2, and t_4 exercises component c_3. The error vector shows that only test t_4 fails, all other tests pass.

[5]Note that code coverage is obtained when running the program with the original test suite, typically in the previous testing iteration.

Algorithm 1 Map Coverage Matrix into Constraints

```
1: Input: Matrix A, an error vector e, number of components M, number of test cases N
2: Output: Conjunctions of disjunctions Ω
3: Ω ← ∅                                          ▷ empty conjunction
4: for all i ∈ {1 ... M} do
5:     Ω' ← ∅                                     ▷ empty disjunction
6:     for all j ∈ {1 ... N} do
7:         if ω(i, j) then
8:             Ω' ← Ω' ∨ j
9:         end if
10:    end for
11:    Ω ← Ω ∧ (Ω')
12: end for
13: for all k ∈ {1 ... N} do
14:    if e(k) then
15:        Ω ← Ω ∧ (k)
16:    end if
17: end for
18: return Ω
```

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \quad \begin{array}{c} e \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

Figure 2: Coverage matrix example with four tests (lines) and three components (columns).

In order to ensure that component c_1 is covered, test t_1 or t_2 needs to be executed. Formally, $(t_1 \vee t_2)$. To cover component c_2, test t_1 or test t_3 are essential, so the mapping is $(t_1 \vee t_3)$. To cover component c_3, tests t_2 and t_4 suffices $(t_2 \vee t_4)$. To maintain the same fault detection, test t_4 (the only one that fails) is also taken into account, so that our approach guarantee the same fault detection (t_4). Thus, the final encoding for this problem is a conjunction of previous constraints

$$\Omega = (\ (t_1 \vee t_2) \wedge (t_1 \vee t_3) \wedge (t_2 \vee t_4) \wedge (t_4)\ )$$

Next section describes how to solve the constraints using a constraint solver.

C. Solving the Constraints

A constraint system comprises a tuple (V, D, Ω) where V is a set of finite variables, D is a function mapping a domain to each variable, and Ω is a finite set of constraints (a conjunction of disjunction) where each constraint has

$$\begin{array}{c} \\ t_1 \\ t_4 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \quad \begin{array}{c} e \\ \begin{bmatrix} 0 \\ 1 \end{bmatrix} \end{array}$$

(a) Set $\{t_1, t_4\}$.

$$\begin{array}{c} \\ t_1 \\ t_3 \\ t_4 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \quad \begin{array}{c} e \\ \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

(b) Set $\{t_1, t_3, t_4\}$.

Figure 3: Coverage matrix example.

a scope (variables from V) and relations restricting the variable values.

Given a constraint system, a Constraint Satisfaction Problem (CSP) finds assignments of values to variables V from their domains D that satisfy constraint Ω. As mentioned before, searching for a solution for Ω is NP-hard in the finite case. However, efficient algorithms for solving the CSPs have been proposed in the past, e.g., [10], [19].

MINION [10] is a general-purpose constraint solver, with an expressive input language based on the common constraint modeling device of matrix models. Experimental results show that MINION is orders of magnitude faster than state-of-the-art constraint toolkits on large and difficult problems [10]. For small problems or instances, where minimal sets are discovered with a simple search, gains are just marginal.

Essentially, the RZOLTAR approach maps the constraint set Ω computed in the previous phase to the MINION language. For the example, the constraint solver returns five possible sets: $\{t_1, t_4\}$, $\{t_2, t_3, t_4\}$, $\{t_1, t_3, t_4\}$, $\{t_1, t_2, t_4\}$, $\{t_1, t_2, t_3, t_4\}$.
The first set, $\{t_1, t_4\}$, minimizes the original test suite in 50%, covering exactly the same code and still maintaining the same fault detection. This test suite is of minimal size because (see Fig. 3a): (i) Not executing t_1 would lead to incomplete coverage, as components c_1 and c_2 are not covered by the other test; (ii) For the same reason, not executing test t_4 would lead to component c_3 being uncovered and would compromise the fault detection rate.
Following the same reasoning, one can easily see that set 2 is also minimal and maintains the same fault detection. Consider set 3 as an example of a non-minimal set: t_3 can be discarded without compromising the code coverage (see Fig. 3b). Sets 4 and 5 (note that this is, in fact, the original test suite) are both valid solutions, but not minimal in term of size.

The fact that the solver computes non-minimal solutions (which are of no interest to us, because we want to provide more than one minimized test suite) is an important limitation that needs to be addressed. The constraint solver yields all possible solutions, and not only those of minimal cardinality. To filter out those solutions that are not minimal in terms of cardinality, we modified MINION to use a TRIE data structure.

A TRIE [20] is an ordered tree data structure used to store a set (set of test cases identifiers in this paper) where the keys are commonly strings. The main idea is that strings with a common prefix share nodes and edges in the tree. The main advantages of the TRIE data structure are: (i) ease with handling sequences of several lengths; (ii) add and/or delete can be easily achieved; (iii) speed of storage and access [21]. A thorough description of the TRIE data structure can be found in [22].

After filtering out the collection of sets yielded by the constraint solver, we would only get $\{t_1, t_4\}$ and $\{t_2, t_3, t_4\}$ as the minimal - in size cardinality - test suites, while still covering all components and maintaining the same fault detection rate as the initial test suite. Once the minimal sets are found and given user input/preferences, the collection is order either using the cardinality or the time needed to execute the minimal set found.

D. Complexity

The worst-time complexity of our approach is $O(|T| + N + 2^M + |R|)$ (execute all test cases; map coverage matrix into constraints; solve constraints; filtering with the TRIE). It is reasonable to assume that $|T|$ and $|R|$ are of the same order of magnitude as N. Thus, time complexity is $O(N + 2^M)$. Since $N \leq 2^M$, the worst-time complexity of our approach is $O(2^M)$, although many problems – such as the ones we try to solve – are in practice solved in polynomial time, $O(M^2)$ [10].

With respect to space complexity, for each invocation of our approach, the complexity is $O(M \cdot N)$ to keep the coverage matrix, $O(|C|)$ to run the constraint solver (where $|C|$ is the number of constraints), $O(T^2)$ for the TRIE to filter out solutions, and $O(|R|)$ to retain all solutions to the problem (where R is the set of solutions). Therefore, the worst-case space complexity is $O(M \cdot N + |C| + T^2 + |R|)$. In practice, however, this complexity did not prove to hinder the applicability of our approach.

IV. EVALUATION

In this section, we evaluate the test suite minimization capabilities of the proposed approach using real software programs and comparing with the state-of-art greedy approach [4]. In particular, we empirically studied the cardinality and execution time reduction of the test suites yielded by our approach and the greedy approach, as well as the time needed to produce the results.

To evaluate the ratio of the size of the minimized test suite from the original one and the ratio of fault detection capability of a minimized test suite, we adopted two measures [23].

To quantify the test suite reduction we use *Reduction in Test-Suite Size* (RS) [23] metric:

$$\text{RS} = \frac{|T| - |T_{minimized}|}{|T|} \times 100 \tag{3}$$

where $|T|$ is the number of tests in original test suite and $|T_{minimized}|$ is the number of tests in the reduced test suite.

To quantify the impact of test suite reduction on the fault detection rate, we use the RF [23] metric:

$$\text{RF} = \frac{|F| - |F_{minimized}|}{|F|} \times 100 \tag{4}$$

Table I: Subject programs detailed.

Subject	Version	Classes	Test Cases	LOCs	Coverage
JMeter	2.6	970	556	84266	34.8%
org.jacoco.report	0.5.7	59	235	2600	97.3%
XML-Security	1.5.0	353	462	24542	64.7%

where $|F|$ is the number of faults in original test suite and $|F_{minimized}|$ is the number of faults in the minimized test suite.

A. Empirical Evaluation

To assess the performance of our approach, we carried out an empirical evaluation using three open source, large software subjects. This study was meant to answer the following research questions.

RQ1: Can RZOLTAR efficiently minimize the original test suite, maintaining the same code coverage and the same fault detection?

RQ2: What is the execution time reduction of RZOLTAR's minimized test suite when compared to the original suite (and the suite computed using the greedy approach)?

We proceed to present the software subjects used in the study and the experimental setup. At the end we report and discuss the results obtained in the experiments.

1) Experimental Subjects: Three subjects, written in Java, were considered in our empirical study. JMeter[6] is a Java desktop application designed to load test functional behavior and measure performance. JaCoCo[7] project's module org.jacoco.report provides utilities for report generation used by JaCoCo itself. XML-Security[8] is a component library implementing XML signature and encryption standards. Both JMeter and XML-Security are sub-projects of the open source Apache project[9].

For each program, we report (see Table I) the version used in our experiments, number of classes, number of JUnit test cases, number of Lines of Code (LOC) (non-comment lines), and percentage of code coverage of the original test suite. Code coverage information was obtained using the open source Eclipse plug-in Metrics[10].

2) Experimental Setup: For all programs, we convert all JUnit test cases in simple unit tests. These programs provide test cases which have at least two or three unit tests. So, to check the real purpose of every unit test, we mapped every unit test into a single test case (e.g. if a test case has 3 unit test (u_1, u_2, and u_3), we create test case c_1 with u_1, c_2 with u_2, and c_3 with u_3). This transformation is valid and legitimate, because it does not change source code, or increase/decrease percentage of coverage or even the number of tests. It is necessary just because of a technological limitation in RZOLTAR: it does not handle the individual tests in a JUnit suite, but considers instead each suite as one test.

[6]JMeter, http://jmeter.apache.org, 2013.
[7]JaCoCo, http://www.eclemma.org/jacoco, 2013.
[8]XML-Security, http://santuario.apache.org, 2013.
[9]Apache, http://www.apache.org, 2013.
[10]Metrics, http://metrics.sourceforge.net, 2013.

Table II: Average ($\bar{t}$) and standard deviation (σ) of time (in seconds) to execute RZOLTAR and greedy approach. The number of minimal test suites provided by each approach is represented by #.

	RZOLTAR			greedy		
Subject	$\bar{t}$	σ	#	$\bar{t}$	σ	#
JMeter	**1.104**	0.009	**2**	16.218	0.099	1
org.jacoco.report	**0.206**	0.007	**1**	0.674	0.004	**1**
XML-Security	**1.525**	0.010	**2**	16.799	0.033	1

For each subject, we repeated the process thirty times to measure the time that RZOLTAR and the greedy approach take to compute the collection of minimal test suites (Eq. 1). We also report the average and the standard deviation σ (detailed in the next section).

The experiments were run on a 2.27Ghz Intel Core i3-350M with 4GB of RAM running Debian Linux Wheezy.

> **RQ1:** Can RZOLTAR efficiently minimize the original test suite, maintaining the same code coverage and the same fault detection?

3) Results and Discussion: The results in Table II show that RZOLTAR can compute two minimal sets of test cases for almost all programs (except for org.jacoco.report) in less than 1.5 seconds, while the greedy approach can only compute - as expected - one set. For instance, for JMeter, with 84266 constraints (number of LOCs) and 556 variables (number of tests cases), the RZOLTAR approach generated two minimal sets in just 1 second, unlike greedy which only return one set in 16 seconds. As can be seen in Table II, for every subjects greedy performed worst than RZOLTAR, 11.9 times on average.

Table III shows the cardinality of the original test suite, the cardinality of one of the minimized test suites yielded by RZOLTAR and greedy, and the reduction in fault detection capability of each approach. We conclude that RZOLTAR can compute two minimal test suites for 2 of the 3 programs considered (except for org.jacoco.report), while greedy can only determine one minimal suite for each subject. The cardinality reduction for the subject programs ranged from 51.98% in case of JMeter, to a significant result of 73.19% in org.jacoco.report. On average, the reduction provide by RZOLTAR is better than the greedy approach. Table III also shows that, greedy has a significant degradation on RF (76.32% on JMeter and 100% on XML-Security). On the other hand, RZOLTAR always maintains the same percentage of fault detection as the original test suite.

Table III: Size of initial test suites $|T|$ of all subject programs, number of tests that P (Pass) or F (Fail), reduction in test-suite size (RS) and reduction in fault-detection capability (RF). Higher RS values denote high reduction, higher RF values denote less fault detection rate.

	Original			**RZOLTAR**					**greedy**				
Subject	$\|T\|$	P	F	$\|T_m\|$	P	F	RS	RF	$\|T_m\|$	P	F	RS	RF
JMeter	556	518	38	267	229	38	51.98%	**0.0%**	255	246	9	**54.14%**	76.32%
org.jacoco.report	235	235	0	63	63	0	**73.19%**	-	66	66	0	71.91%	-
XML-Security	462	461	1	141	140	1	**69.48%**	**0.0%**	167	167	0	63.85%	100.0%

Table IV: Average time ($\bar{t}$) in seconds needed to execute the original test suite, the reduced proposed by RZOLTAR and greedy approach, and the % of time reduction afford by each approach.

	Original	**RZOLTAR**		**greedy**	
Subject	$\bar{t}$	$\bar{t}$	%	$\bar{t}$	%
JMeter	28.844	24.356	15.56%	23.878	**17.22%**
org.jacoco.report	3.423	1.206	**64.77%**	1.627	52.47%
XML-Security	30.056	13.094	**56.43%**	18.089	39.82%

RQ2: What is the execution time reduction of RZOLTAR's minimized test suite when compared to the original suite (and the the greedy approach)?

The time needed to execute the computed, minimal test suite with RZOLTAR and greedy was also measured, and compared it to the original suite.
As expected, when reducing the size of the original test suite, one reduces the time needed to achieve the same coverage. Table IV shows the time needed to execute the original suite and the reduced one (minimal set) proposed by RZOLTAR and greedy. Similar to the results reported for the cardinality, the time reduction for minimal sets calculated by RZOLTAR is on average better then the greedy approach. For instance, for XML-Security subject RZOLTAR reduces the execution time in 69.48% and greedy 63.85%.

B. Threats to Validity

Despite the programs used in the empirical results are real, large and open source software, the main threat to the external validity is the fact only three subjects were used. It is plausible to conclude that the results for a different set of programs, with different characteristics, may generate different results.

To mitigate this threat, we have not only thoroughly tested the toolset but also manually checked a large set of results.

V. RELATED WORK

Trying to find the minimal test suite (Eq. 1) that covers the same set of requirements as the original one is a NP-hard problem, but can be solved in a polynomial time using the minimal hitting set problem [8]. NP-hardeness of the test suite minimization problem encourages the usage of heuristics. Precedent work on test case minimization has advanced the state-of-the-art of heuristic approaches to the minimal hitting set problem [1], [2], [3], [4], [5], [6], [7].

Chavatal [1] proposes the usage of a greedy heuristic (i.e., an approximation) to efficiently solve the minimization problem. This greedy approach selects test cases until all requirements have been covered, but without guaranteeing that only the test cases needed are added to the suite. First, it selects the test case that covers the large number of requirements. If there are more than one test that covers the same number of requirements, one is randomly selected. The approach keeps doing this selection until all requirements in the original suite have been covered. One limitation of such heuristic-based approach is that the selection of a test case may potentially be redundant by the test cases that will be selected later in the process.

Offutt, Pan, and Voas (OPV for short) [2] proposed an approach that is a variation of the greedy heuristic approach just outlined. Their approach differs from the original greedy because they use multiple heuristics to elect the test cases. Regardless, and similar to [1], this approach still yields an approximation, with no guarantees of computing the optimal solution.

Another greedy heuristic has been developed by Harrold, Gupta, and Soffa (HGS for short) [3]. Their heuristic is based on discarding obsolete and redundant test cases from the original test suite. This heuristic approach computes a minimized test suite which has the same size or is smaller than the ones computed by Chavatal [1], but entails a worst time execution. Like OPV, the solution computed by this approach are more efficient than Chavatal [1]. No assurance given in terms of fault detection rate given by the yielded test suite. Chen *et al.* [6] developed an HGS-based approach that takes into account the interaction between test requirements in order minimize the test suite and to improve the fault detection rate.

Tallam and Gupta [4] developed an approach, coined Delayed-Greedy, that tackles the weaknesses of the greedy approach. Delayed-Greedy works in three steps: (i) it discards test cases that cover some requirements covered by other test cases; (ii) it discards test requirements which are not part of the minimized requirements set; and (iii) it uses the classical greedy approach to create a minimized test suite with the remainder test cases. An empirical evaluation [4]

has shown that this approach computes minimized test suites of the same size or smaller than the classical greedy and the HGS approaches.

More recently Hao, Zhang, Wu, Mei and Rothermel [5] introduced a new technique that first checks individual statements and stores statistics about the possibility of each statement losing capability in fault-detection; second models the minimization as an Integer Linear Programming (ILP) problem.

Lin, Tang, Chen and Kapfhammer [7] proposes an algorithm that takes into account the ratio of coverage to time of test cases as the minimization criteria. On their experiments, only using the rather small Siemens benchmark suite, they concluded that this approach does not scale to large programs and test suites.

To our knowledge, our approach is the first to leverage a constraint solver to generate multiple optimal test suites, each with the same coverage as the original suite.

VI. Conclusions and Future Work

In this paper, a new approach, dubbed RZoltar, for test suite minimization was proposed. It takes as input the requirements covered by the test cases in the suite (code coverage in the context of this paper) and, using a constraint solver programming approach, minimizes the suite, while still guaranteeing that the testing requirements are met. The collection of generated suites can then be ranked by the user (e.g., by cardinality or time needed to execute). To facilitate the adoption of our approach, we have integrated it within the GZoltar Eclipse plug-in [11].

Application to three real-world, open source, and large programs indicates that RZoltar can significantly reduce the original test suite, while still maintaining the full code coverage and the fault detection rate. We observed averaged reductions of 64.88% in terms of test suite size and 45.59% of execution time reduction (when compared to the original one). RZoltar was also compared with the greedy approach, and on average yielded better results in term of reduction of all test suites and time reduction.

Our approach is also better in terms of fault detection. RZoltar has a RF of 0.0% for every subject, which means that it guarantees the same fault detection as the initial test suite. On the other hand, greedy approach has a significant reduction on RF, e.g, 76.32% for JMeter.

Future work includes the following. We plan to investigate techniques to prioritize the execution of the test cases in the reduced test suite to enhance the rate at which diagnostic quality improves.

Acknowledgment

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology within project FCOMP-01-0124-FEDER-020484.

References

[1] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[2] J. Offutt, J. Pan, and J. Voas. Procedures for reducing the size of coverage-based test sets. In *Proc. of the 12th ICTCS'95*.

[3] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[4] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *SIGSOFT Softw. Eng. Notes*, 31(1):35–42, September 2005.

[5] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *Proc. of the ICSE '12*, pages 738–748, 2012.

[6] Xiang Chen, Lijiu Zhang, Qing Gu, Haigang Zhao, Ziyuan Wang, Xiaobing Sun, and Daoxu Chen. A test suite reduction approach based on pairwise interaction of requirements. In *Proc. of the SAC '11*, pages 1390–1397, 2011.

[7] Chu-Ti Lin, Kai-Wei Tang, Cheng-Ding Chen, and Gregory M. Kapfhammer. Reducing the Cost of Regression Testing by Identifying Irreplaceable Test Cases. In *Proc. of the 6th ICGEC '12*.

[8] R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W. H. Freeman, 1979.

[9] Staal Vinterbo and Aleksander Øhrn. Minimal approximate hitting sets and rule templates. *International Journal of Approximate Reasoning*, 25(2):123 – 143, 2000.

[10] Ian P. Gent, Chris Jefferson, and Ian Miguel. MINION: A Fast, Scalable, Constraint Solver. In *Proc. of the 17th ECAI' 06*, pages 98–102, 2006.

[11] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proc. of the 27th ASE '12*, pages 378–381, 2012.

[12] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, 2009.

[13] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[14] A.G. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proc. of the ICSM '02*, pages 204 – 213, 2002.

[15] H.P. Williams. *Model building in mathematical programming*. Wiley-Interscience publication. Wiley, 1985.

[16] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987.

[17] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artif. Intell.*, 41(1):79–88, November 1989.

[18] Franz Wotawa. A variant of Reiter's hitting-set algorithm. *Inf. Process. Lett.*, 79(1):45–51, May 2001.

[19] Dharini Balasubramaniam, Christopher Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. An automated approach to generating efficient constraint solvers. In *Proc. of the ICSE '12*, pages 661–671, 2012.

[20] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960.

[21] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proc. of the 22nd AAAI'07*.

[22] K.D. Forbus and J. De Kleer. *Building Problem Solvers*. Number v. 1 in Artificial Intelligence. Mit Press, 1993.

[23] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. An Empirical Study of JUnit Test-Suite Reduction. In *Proc. of the 22nd ISSRE '11*, pages 170–179, 2011.