Encoding Test Requirements as Constraints for Test Suite Minimization

José Campos, Rui Abreu
Faculty of Engineering of University of Porto, Portugal

Abstract

Software (regression) testing is performed not only to detect errors as early as possible but also to guarantee that changes did not affect the system negatively. As test suites tend to grow over time, e.g., new test cases are added to test new features, (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 set of optimal solutions (maintaining the same coverage as the original suite) by leveraging a fast constraint solver. We show that* RZOLTAR *efficiently (0.68 seconds on average) finds a collection of test suites that significantly reduce the size of the original suite (61.12%), while greedy only finds one solution with a reduction of 56.58% in 6.92 seconds on average.*

Keywords: Constraint solver, empirical evaluation, regression testing, test suite reduction.

1. Introduction

In recent years, the software testing research community has given considerable attention to the subject of regression testing. Some approaches [4, 6, 12, 13, 14, 15, 16, 18], 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) for time efficiency (e.g., [6, 16]).

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 [9]. Despite being a NP-complete problem [9, 19], 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 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 [6]);

(ii) It produces a collection of solutions (and not only just one solution like the greedy [6] or the integer programming approach [14]).

As our approach yields more than one solution, 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 [22]. To facilitate the adoption of our technique, we implemented RZOLTAR within the GZOLTAR toolset [5]. GZOLTAR is an Eclipse[1] plug-in for automatic debugging, offering graphical visualizations of diagnostic reports produced by spectrum-based fault localization [2]. The main reason for this choice is to allow the developers to adopt the tool without much effort as it is already integrated in the Eclipse IDE and takes as input JUnit[2] 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 61.12% in terms of number of test cases and 44.49% of time reduction, while still maintaining full code coverage. Comparing with the simple greedy approach [6] which is amongst the best performing heuristics, RZOLTAR reduces the size of the original suite more than greedy, and provides more than one solution for almost all open-source projects used in the evaluation in less time (10.23 times 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;
- The proposed technique has been implemented within the GZOLTAR toolset [5], more specifically in a cutting-edge Eclipse view dubbed RZOLTAR, this way providing an ecosystem for testing and debugging software programs;

[1] The Eclipse Foundation website, http://www.eclipse.org/, 2013.
[2] JUnit's official website, http://www.junit.org/, 2013.

- We empirically evaluate the test minimization capabilities of RZOLTAR using large, real world software programs;
- We compare the performance and results of our approach with greedy, known as an effective time algorithm [22].

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

2. Motivating Example

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 m_1, line 2 is component m_2, and line 3 is component m_3). The value $M = |\{m_1, m_2, m_3\}|$ represent the number of components in the example program. During the development phase of the software development life-cycle it is usually the case to have a set of testing requirements for Π. Note that it is irrelevant for our approach what requirements are, but in this paper we assume statement coverage.

Definition 1 *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 works as expected or not, and o is the expected output. If $\Pi(i) = o$ the test case passes, otherwise fails.*

Definition 2 *A test suite $T = \{t_1, \ldots, t_N\}$ is a collection of test cases that are intended to test whether the program follows the specified set of requirements. The cardinality of T is the number of test cases in the set $|T| = N$.*

As an example consider the following test suite $T = \{t_1, t_2, t_3, t_4\}$. The 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)`, test case t_3 checks `sub(1,0)`, and test case t_4 checks `mul(0,1)`.

During the development phase of software, it is common practice to build and maintain a *regression test suite*. 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 *and* t_4 already checks all requirements.

```
   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 achieving the same code coverage as in the original suite, consider that the testing requirements for each test case can be encoded as a set. For example, $\{m_1, m_2\}$ are the testing requirements for t_1. Taking as input a collection of sets, one per test case, we would like to find the minimal hitting 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[3] of the collection of covered requirements set.

Definition 3 *Minimal hitting set is the problem of finding the minimal sub collection (cover) of subsets whose union gives elements that cover all the elements of main set.*

Identifying the minimal hitting set of a collection of sets is an important problem in many domains (e.g., such as Air Crew Scheduling [20]). Being an NP-complete problem (i.e., exponential on the number of components) [9], brute-force algorithms are prohibitive for real-world, often large programs. Furthermore, heuristic approaches (e.g., STACCATO [1], greedy [6]) trade-off completeness for time efficiency. In the next section, we present 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 solutions for the minimization problem. As multiple solutions are generated, the developer can prioritize them using two criteria: (i) cardinality of the minimized test suite or (ii) test suite's execution time.

3. 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 test suites that guarantee both the same coverage 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 3.1);

[3] A minimal hitting set is a hitting set such that none of its subsets is a hitting set.

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 3.2);

3. The constraints are solved with the slightly modified, off-the-shelf constraint solver MINION, and prioritized using a certain criterion (detailed in Section 3.3).

We now detail each of these phases.

3.1. 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[4].

Each test case in the test suite, when run through the program, will cover a subset of the total set of components (statements in the context of this paper). By tracking which components each test case activate (covers), a $N \times M$ binary matrix A is created. We refer to this matrix as *coverage matrix*. An element a_{ij} is equal to 1 if and only if component m_j is covered when the test t_i is executed:

$$\omega(t_i,\ m_j) = (a_{ij} == 1\ ?\ true : false) \qquad (1)$$

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 $\omega(t_i,\ m_j)$ function in Eq. (1), 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 explore the information in the coverage matrix to minimize and prioritize the current test suite.

3.2. 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.

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \end{array} \begin{array}{c} \begin{array}{ccc} m_1 & m_2 & m_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \end{array}$$

Figure 2. Coverage matrix example, with four tests (lines) and three components (columns). E.g., the component m_2 is covered when the test t_1 or t_3 is executed.

The key idea behind our approach is to encode the testing requirement of each test case into a set of constraints, 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: m_1 correspond to statement with number `1`, m_2 to statement `2`, and m_3 to statement `3`. The four test cases execute at least one component: t_1 exercises components m_1 and m_2, t_2 covers only component m_1, t_3 exercises component m_2, and t_4 exercises component m_3.

In order to ensure that component m_1 is covered, test t_1 or t_2 needs to be executed. Formally,

$$(t_1 \vee t_2)$$

To cover component m_2, test t_1 or test t_3 are essential, so the mapping is

$$(t_1 \vee t_3)$$

To cover component m_3, test t_4 suffices

$$(t_4)$$

Thus, the final encoding for this problem is (a conjunction of previous constraints) is

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

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

3.3. Solving the Constraints

A constraint system comprises a tuple (V, D, C) where V is a set of finite variables, D is a function mapping a domain to each variable, and C is a finite set of constraints where each constraint has 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 C. As mentioned before, searching for a solution for C is NP-complete in the finite case. However, efficient algorithms for solving the CSPs have been proposed in the past, e.g., [3, 10].

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 solutions are discovered with a simple search, gains are just marginal. Due to lack of space, we do not detail how the model of Section 3.2 is encoded into MINION.

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

[4]Note that code coverage is obtained when running the program with the original test suite.

TRIE [8] 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 [11]. A thorough description of the TRIE data structure can be found in [7].

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 solutions, while still covering all components. Once the solutions are found and given user input/preferences, the collection is order either using the cardinality or the time needed to execute the minimal solutions found.

3.4. Tooling

The lack of tools offering state-of-the-art techniques for test suite minimization impairs wide adoption, namely by industry. For instance, to perform unit testing there are a number of frameworks based on the xUnit architecture, but, to our knowledge, only a few really provide support for testing minimization [22]. Seeking wide adoption of our technique, we integrated the proposed technique into an Eclipse view, coined RZOLTAR, in the GZOLTAR [5] Eclipse plug-in, available online at:

`http://www.gzoltar.com`

4. Evaluation

In this section, we evaluate the test suite minimization capabilities of the proposed approach using real software programs and comparing it with the state-of-art greedy approach [6]. 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.

4.1. Empirical Evaluation

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

RQ1: Can RZOLTAR efficiently minimize the test suite, maintaining the same code coverage?

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)?

Sections 4.1.1 and 4.1.2 present the software subjects used in the study and the experimental setup, respectively. Section 4.1.3 reports and discusses the results obtained in the experiments.

Table 1. Subject programs detailed.

Subject	Version	Classes	Test Cases	LOCs	Coverage
JMeter	2.6	970	556	84266	34.8%
JTopas	0.8	57	160	4373	71.9%
NanoXML	2.2.3	29	9	4660	56.2%
org.jacoco.report	0.5.7	59	235	2600	97.3%
XML-Security	1.5.0	353	462	24542	64.7%

4.1.1. Experimental Subjects. Five subjects, written in Java, were considered in our empirical study. JMeter[5] is a Java desktop application designed to load test functional behavior and measure performance. JTopas[6] is a Java library used for parsing text data. NanoXML[7] is a small XML parser for Java. JaCoCo[8] project's module org.jacoco.report provides utilities for report generation used by the JaCoCo itself. XML-Security[9] is a component library implementing XML signature and encryption standards. Both JMeter and XML-Security are sub-projects of the open source Apache project[10].

All programs, except NanoXML, provide test suites in JUnit. NanoXML's test suite is encoded in a Test Script Language (TSL) suite [17]. Basically, in TSL, tests are defined by an input/output file containing the input/output pair per test.

For each program, we report (see Table 1) 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[11].

4.1.2. Experimental Setup. As NanoXML does not provide JUnit tests and our toolset takes as such tests only (the only format supported by RZOLTAR at the moment), we have converted the TSL tests into JUnit tests. Essentially, each input/output pair in the TSL test suite is mapped into a JUnit test which checks whether the output holds for a given input. Hence, we still maintain inputs/outputs of tests and the purpose of the testing pair in the TSL suite.

For all the other 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

[5] JMeter, `http://jmeter.apache.org`, 2013.
[6] JTopas, `http://jtopas.sourceforge.net`, 2013.
[7] NanoXML, `http://devkix.com`, 2013.
[8] JaCoCo, `http://www.eclemma.org/jacoco`, 2013.
[9] XML-Security, `http://santuario.apache.org`, 2013.
[10] Apache, `http://www.apache.org`, 2013.
[11] Metrics, `http://metrics.sourceforge.net`, 2013.

Table 2. Cardinality of original suite ($|T|$) and minimized test suite ($|T_m|$) % of reduction, for every subject, provided by RZOLTAR and greedy approach.

	Original	RZOLTAR		greedy	
Subject	$\|T\|$	$\|T_m\|$	%	$\|T_m\|$	%
JMeter	556	**237**	**57.37%**	255	54.14%
JTopas	160	**27**	**83.13%**	29	81.88%
NanoXML	9	**7**	**22.22%**	8	11.11%
org.jacoco.report	235	**63**	**73.19%**	66	71.91%
XML-Security	462	**140**	**69.70%**	167	63.85%

tests in a JUnit suite, but considers instead each suite as one test.

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

The experiments were run on a 2.27 Ghz Intel Core i3-350M with 4 GB of RAM running Debian GNU/Linux Wheezy[12].

4.1.3. Results and Discussion.

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

Table 2 plots the cardinality of the original test suite, the cardinality of the minimized test suite yielded by RZOLTAR and greedy. The cardinality reduction for the subject programs ranged from 22.22% in case of NanoXML, to a significant result of 83.13% in JTopas. For JMeter the reduction was about 57.37%, org.jacoco.report 73.19%, and XML-Security 69.70%. For the greedy approach, the reduction for JMeter was about 54.14%, JTopas 81.88%, NanoXML 11.11%, org.jacoco.report 71.91%, and XML-Security 63.85%.

The results in Table 3 show that RZOLTAR can compute more than one minimal sets of test cases for almost all programs (except for org.jacoco.report) in less than 3 seconds, and in most cases the solutions were computed even in less than 0.5 seconds. On the other hand, greedy can only determine one solution for each subject. 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 returns one set in 16 seconds. As we can see in Table 3, for every subjects greedy performed worst than RZOLTAR, 10.23 times on average.

[12]Debian GNU/Linux, `http://www.debian.org`, 2013.

Table 3. Average and standard deviation (σ) of time (t) (in seconds) to execute RZOLTAR and greedy approach. And the number (#) of minimized test suites.

	RZOLTAR			greedy		
Subject	t	σ	#	t	σ	#
JMeter	**1.115**	0.027	**2**	16.190	0.076	1
JTopas	**0.475**	0.015	**2**	0.725	0.004	1
NanoXML	**0.042**	0.004	**2**	0.169	0.005	1
org.jacoco.report	**0.205**	0.004	**1**	0.679	0.007	**1**
XML-Security	**3.046**	0.066	**3**	16.852	0.034	1

Table 4. Average time (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	t	t	%	t	%
JMeter	28.844	23.405	**18.86%**	23.878	17.22%
JTopas	2744.891	852.067	68.96%	836.914	**69.51%**
NanoXML	0.417	0.361	**13.43%**	0.374	10.31%
org.jacoco.report	3.423	1.206	**64.77%**	1.627	52.47%
XML-Security	30.056	13.092	**56.44%**	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 suite computed using the greedy approach)?

We also measured the time needed to execute the minimal test suite with RZOLTAR and greedy, 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 4 shows the time needed to execute the original set and the reduced one (minimal set) proposed by RZOLTAR and greedy. Similar to the results reported for the cardinality (where RZOLTAR obtains better results), the time reduction for minimal sets calculated by RZOLTAR is (in most cases) greater than the greedy approach. For instance, for XML-Security subject RZOLTAR reduces the execution time in 56.44% and greedy only in 39.82% (on average).

4.2. 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 five subjects were used. It is plausible to conclude that the results for a different set of programs, with different characteristics, may generate different results.

Threats to internal validity revolves around eventual faults in the RZOLTAR implementation or even in the underlying constraint solver. To mitigate this threat, we have not only thoroughly tested the toolset but also manually checked a large set of results.

5. Related Work

Trying to find the minimal test suite that covers the same set of requirements as the original one is a NP-complete problem, but can be solved in a polynomial time using the minimal hitting set problem [9]. The NP-completeness 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 [4, 6, 12, 13, 14, 15, 16, 18]. Due to space limitations, we refrain from detailing the approaches.

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.

6. 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 [5].

Application to five real-world, open source, and large programs indicates that RZOLTAR can significantly reduce the original test suite, while still maintaining the full code coverage. We observed averaged reductions of 61.17% in terms of test suite size and 63.98% of execution time reduction. RZOLTAR was also compared with the greedy approach, and yielded better results in term of reduction of all test suites and time reduction.

Future work includes the following. We will investigate the fault detection capabilities of the reduced test suites (similar to Wong *et al.* work [21]), as even though they achieve the same coverage, the reduction may have a negative impact in fault detection.

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 PTDC/EIA-CCO/116796/2010.

References

[1] Rui Abreu and Arjan J. C. van Gemund. A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. In *Eighth Symposium on Abstraction, Reformulation, and Approximation*, SARA '09. AAAI, 2009.

[2] 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, November 2009.

[3] Dharini Balasubramaniam, Christopher Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. An automated approach to generating efficient constraint solvers. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 661–671, Piscataway, NJ, USA, 2012. IEEE Press.

[4] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-Criteria Models for All-Uses Test Suite Reduction. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 106–115, 2004.

[5] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: An Eclipse Plug-In for Testing and Debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 378–381, 2012.

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

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

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

[9] 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.

[10] Ian P. Gent, Chris Jefferson, and Ian Miguel. MINION: A Fast, Scalable, Constraint Solver. In *Proceedings of the 17th European Conference on Artificial Intelligence*, pages 98–102, 2006.

[11] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, AAAI'07, pages 191–197, 2007.

[12] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 738–748, 2012.

[13] 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.

[14] Hwa-You Hsu and Alessandro Orso. MINTS: A General Framework and Tool for Supporting Test-suite Minimization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 419–429, 2009.

[15] Dennis Jeffrey and Neelam Gupta. Test Suite Reduction with Selective Redundancy. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 549–558, 2005.

[16] J. Offutt, J. Pan, and J. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the Twelfth International Conference on Testing Computer Software*, June 1995.

[17] T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Commun. ACM*, 31(6):676–686, June 1988.

[18] 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.

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

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

[21] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, 1998.

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