A Multi-Objective Technique for Test Suite Reduction

Alessandro Marchetto, Md. Mahfuzul Islam, Angelo Susi
Fondazione Bruno Kessler
Trento, Italy
{marchetto,mahfuzul,susi}@fbk.eu

Giuseppe Scanniello
Università della Basilicata
Potenza, Italy
giuseppe.scanniello@unibas.it

***Abstract*—Entire test suites are often used to conduct regression testing on subject applications even after limited and precise changes performed during maintenance operations. Often, this practice makes regression testing difficult and costly. To deal with these issues, techniques to reduce test suites have been proposed and adopted. In this paper, we present a multi-objective technique for test suite reduction. It uses information related to the code and requirements coverage, the past execution cost of each test case in the test suite, and traceability link among software artifacts. We evaluated our proposal by testing three Java applications and comparing the achieved results with those of some baseline techniques. The results indicate that our proposal outperforms the baselines and that improvements are still possible.**

Keywords*—*Regression Testing; Requirements; Testing; Test Suite Reduction; Traceability Link Recovery.

I. INTRODUCTION

Regression testing is usually conducted after software maintenance operations to guarantee that the effect of these operations does not compromise the expected behavior of a software application. Relevant activities often conducted during regression testing [1] are: *(i)* test selection; *(ii)* test reduction (also named minimization); and *(iii)* test prioritization. These activities are technical and business relevant because they might affect the success of a software project [2]. Among the activities above, test reduction reduces the number of test cases to be executed and should preserve the capability of a test suite in discovering faults.

To reduce test suites, existing techniques are mostly based on a single dimension (e.g., code or requirements coverage). Few attempts exist to reduce test suites and apply multiple dimensions only considering *structural* information (e.g., code coverage and execution cost), thus ignoring the *functional* dimension [3][4]. Conversely, it could be relevant to reduce test suites by explicitly taking into account structural and functional information, and the time (e.g., seconds) required to execute them.

In this paper, we propose a novel reduction technique named MORE (Multi-Objective test cases REduction). It is multi-objective and selects a subset of a test suite (i.e., reduced test suite), so decreasing the testing time while preserving the capability of the suite in exercising the application and detecting faults. The technique is based on a three-dimension analysis of test cases. The *structural* dimension concerns information regarding test cases under analysis (i.e., how they exercise the application under test), while *functional* dimension regards the coverage of users' and system requirements. The last dimension is *cost* and concerns the time to execute test cases. To deal with these dimensions traceability links among software artifacts (i.e., application code, test cases, and requirements specifications) are needed. Traceability links are often not available or not up-to-date in the project documentation. Then, we exploit Latent Semantic Indexing (LSI) [5] to infer traceability links among software artifacts and to measure their strength. To assess the validity of MORE, we have conducted an experimental evaluation on three Java applications. In this evaluation, we were mainly interested in assessing whether the test suite reduced by applying our proposal may be effective and efficient as the entire test suite.

Structure of the paper. In Section II, we discuss related work, while the used traceability recovery approach is described in Section III. In Section IV, we highlight the approach for test suite reduction, while the experiment is presented in Section V. Final remarks and future work conclude.

II. RELATED WORK

The greater part of the approaches for test suite reduction is single-objective, [1][3]. However, multi-objective techniques have been also proposed. They largely adopt evolutionary algorithms by reformulating the test suite reduction problem as an optimization problem [15][16][17]. These approaches consider either code or requirement coverage information and try balancing that information with the execution cost of test cases as follows: *(i)* explicitly optimize them as two objectives (e.g., code coverage and execution cost); *(ii)* redefine the multi-objective to a single-objective by using an optimization function that conflates more objectives into only one. For instance, Yoo *et al.* [15] showed the benefits of the Pareto-front optimality respectively for test case selection and test minimization. They, in fact, present a two-objective approach in which code coverage and execution cost are explicitly considered when conducting test selection or minimization. To reduce test suites, MA *et al.* [17] adopted an objective function that conflates code coverage and execution cost information. Furthermore, de Souza *et al.* [16] proposed the use of the Particle Swarm Optimization (PSO) algorithm that considers two objectives for test case selection: coverage of functional requirements and execution cost.

Differently from the paper discussed above, we propose a technique to reduce test suites by explicitly considering both low- (e.g., code coverage) and high-level (e.g., requirements coverage) information about the test cases, as well as their execution cost. We fill the gap between these kinds of information by using LSI [5] to automatically recover traceability links among software artifacts. Moreover, conversely to our previous work [18], we investigated the problem of reducing large test suites and, to this aim, we formulated the problem as

 ISBN: 978-1-61208-304-9

a multi-objective optimization problem and adopted a specific implementation of the NSGA-II algorithm [9].

III. TRACEABILITY RECOVERY

In this paper, we applied an IR-based technique to recover traceability links: *(i)* among high-level software artifacts (i.e., application requirements and test case specifications) and low-level software artifacts (i.e., source code of the application and test case implementations); and *(ii)* between pairs of high-level software artifacts (i.e., application requirements). We use here *textual representations* of these artifacts. In the case of the test cases (implemented using special conceived frameworks, e.g., Junit), a preliminary analysis was performed to identify the application code identifiers (e.g., method and attribute names) executed by test cases. The identifiers will constitute the textual representation of the test cases. We used here LSI [5] as the IR technique. The motivation for using LSI is that it has been successfully used in the traceability recovery field [6].

A. LSI and IR-Based Traceability Recovery

LSI assumes that there is some underlying or "latent structure" in word usage that is partially obscured by variability in the word choices. To this end, a Singular Value Decomposition (SVD) is applied to a $m \times n$ matrix (also named term-by-document matrix), where m is the number of terms, and n is the number of documents in the collection. SVD can be geometrically interpreted: each term and artifact could be represented by a vector in the k space of the underlying concepts. In traceability recovery field, the similarities between two documents or between a term and a document are computed using the cosine between the vectors in the latent structure. In this work, we applied this similarity measure. The larger the value, more similar the vectors are. A value for k should be large enough to fit all the real structure in the data, but small enough so that we do not also fit the sampling error or unimportant details [5]. As default value, we used *k*=300.

Differently from typical text retrieval problems (a user writes a textual query and documents that are similar to the query are shown), in IR-based traceability recovery a set of source code artifacts (used as the query) are compared with a set of target artifacts (even overlapping). Candidate traceability links (i.e., all the possible pairs of software artifacts) are reported in a ranked list. Irrelevant links are removed using a threshold that selects only retrieved links (a subset of top links). In this work, we use the *Constant Threshold* method: 0.1 is the default value used. We used this value to limit the possibility of loosing links by considering a larger number of possible traceability links. There are also methods that do not take into account the similarity values between source and target software artifacts. For example, the method *Variable Cut Point* requires the specification of the percentage of links of the ranked list to be considered as correctly retrieved. Either relevant traceability links could be lost or irrelevant traceability links could be introduced by using methods not based on similarity values.

As in traditional IR-based traceability recovery approaches, our solution retrieves links that are either correct or incorrect so needing the human intervention to remove erroneously recovered links. To avoid that human factors may affect the experimental results, we did not perform here any analysis on the recovered links.

IV. TEST SUITE REDUCTION

We introduce our technique and the metrics used.

- **Code**. The fault detection capability of a test case and then of a test suite represents the capability to detect faults in source code. This cannot be known before executing test cases. Then, we have to resort to the "potential" fault detection capability of a test suite. It can be estimated considering the amount of code covered by test cases. A test case that covers a larger set of code statements at run-time has a higher potential fault detection capability (i.e., more faults should be revealed) than one test case that covers a smaller set of statements.

Assuming to have test case implementations (e.g., Junit test cases), we define *CCov(t)* as the amount of statements exercised during the implementation *t*:

$$CCov(t) = \sum_{s \in Statements} \begin{cases} 1 & s \in CodeCovered \\ 0 & otherwise \end{cases} \quad (1)$$

where *Statements* is the set of source code statements. *CodeCovered* is the set of statements covered by the execution of the test case *t*, *s* is a code statement of the application. Given a test suite *S* composed of ordered test cases, we defined *cumCCov(t_i)* as follows:

$$cumCCov(t_i) = \sum_{j=0}^{i-1} CCov(t_j) \quad (2)$$

where t_i is a test case of the suite. The cumulative code coverage for t_i is computed by summing the single code coverage (i.e., the code covered only by the test case) of all those test cases from t_0 to t_{i-1}.

- **Requirements**. The capability of a test case to exercise users' and/or system requirements depends on: *(i)* the amount of the requirements covered by the test case; *(ii)* the relevance of the covered requirements; and *(iii)* the existing dependency/relationship among requirements. We defined and used *RCov(t)* and a weighted variant *WRCov(t)*. *RCov(t)* is the measure of the requirements coverage for the test case *t*. This measure estimates the application requirements exercised during the execution of *t* and it is computed by counting the number of requirements exercised by the test case *t*. *WRCov(t)* measures the coverage for a test case according to predefined weights assigned to each application requirement. This coverage measure is computed as follows:

$$WRCov(t) = \sum_{r \in Reqs} \begin{cases} w_r & r \in ReqsCovered \\ 0 & otherwise \end{cases} \quad (3)$$

Reqs is the set of requirements of the application under test. *ReqsCovered* is the set of requirements covered by the execution of the test case *t*, while *r* is one of the application requirement and w_r ($0 \leq w_r \leq 1$) is the predefined weight associated to each requirement. Notice that if we consider all requirements equally (i.e., w_r=1), we resort to *RCov(t)*. The requirements weight w_r depends on the testing needs. In this

work, we use as default values three weights associated to the labels *high*, *medium*, *low* [2]:

$$w_r = \begin{cases} 1 & r \in TesterRelevant_r \\ 0.5 & TesterPartialRelevant_r \\ 0 & TesterNonRelevant_r \end{cases} \quad (4)$$

where *TesterRelevant*$_r$ and *TesterPartialRelevant*$_r$ are those requirements *r* selected by the tester as relevant or partially relevant, instead the remaining requirements are *TesterNonRelevant*$_r$. However, alternative definition of the weight w_r can be considered. In fact, as in the code coverage case, the use of this weight w_r is expected to be useful to customize the measurement of the requirements coverage according to the tester's need. Hence, requirements prioritization techniques [7] could be applied to automatically identify requirements that are relevant for the tester's purposes and then to be highly weighted when measuring the coverage.

RCov(t) and *WRCov(t)* do not consider the existing relationship among requirements: all the requirements are considered equally. This issue can lead to situations in which groups of slightly connected requirements (i.e., those requirements having a limited number of related requirements) are privileged than the more connected ones. To deal with this issue, we define *WRCovD(t)*. It takes into account existing relationships among requirements. For sake of simplicity, in the following, we applied the variant only to *WRCov(t)* but the same could be done with *RCov(t)*. To compute *WRCovD(t)*, we need to measure the strength of each requirements relationship/dependency (rD). This strength is computed as follows:

$$w_{rD}(r_l, r_m) = \frac{w_{req}(r_l, r_m) + w_{code}(r_l, r_m)}{2} \quad (5)$$

where $w_{rD}(r_l, r_m)$ is the weight of the relationship in rDs between requirements: r_l and r_m; $w_{rD}(r_l, r_m)$ tends to 1 if a strong relationship exists between r_l and r_m, i.e., both textual description and implementation strongly overlap, while $w_{rD}(r_l, r_m)$ tends to 0 if no relationship exists between r_l and r_m. $w_{req}(r_l, r_m)$ and $w_{code}(r_l, r_m)$ are the weights of the relationship with respect to requirements r_l and r_m and their implementation code, and are computed as follows:

$$w_{req}(r_l, r_m) = IRSimilarity(r_l, r_m) \quad (6)$$

$$w_{code}(r_l, r_m) = \frac{overlapClasses(r_l, r_m)}{totalClasses(r_l, r_m)} \quad (7)$$

$w_{req}(r_l, r_m)$, inferred by LSI, provides an indication about the possible link between the application requirements r_l and r_m, while $w_{code}(r_l, r_m)$ computes the portion of code that is in common between the implementation of the requirements r_l and r_m.

The final requirement coverage of *t* is computed as:

$$WRCovD(t) = \sum_{r \in Reqs} w_r * (\sum_{r_l \neq r \in Reqs} w_{reqs}(r, r_l)) \quad (8)$$

where w_r is the predefined weight associate to each requirement. The weight of the dependencies between the current requirement r and the other requirements of the application are computed by the formula: $\sum_{r_l \neq r \in Reqs} w_{rD}(r, r_l)$. $WRCovD(t_i)$ is expected to give more relevance than *WRCov(t)* to the test cases covering requirements having strong relationships with a high number of other requirements, that is to the test cases exercising "key" requirements.

Given a test case $t_i \in S$, we define:

$$cumRCov(t_i) = \sum_{j=0}^{i-1} WRCovD(t_j) \quad (9)$$

The cumulative requirements coverage for the test case t_i is computed by summing the single requirements coverage (i.e., the requirements covered only by the test case) of all those test cases from t_0 to t_{i-1}.

- **Execution cost**. The execution cost of a test case can be approximated by the time required to its execution. If the implementation of the test cases is available, their execution can be profiled to collect the information about the running time. Alternatively, we can approximate the execution time by counting the number of software elements (e.g., code classes, methods) expected to be exercised by the test case. In this work, we assume to have the test implementation (e.g., Junit test cases), thus we defined *Cost(t)* as the estimated time required to execute the test case.

Therefore, given a test suite *S*, whose test cases are ordered, we computed *cumCost(t_i)* as the sum of the execution costs of the test cases preceding the test case $t_i \in S$. The overall cost of the test cases of a suite *S* (named *Cost(S)*) is the sum of the executions of all the test cases. We then define *InverseCost(t_i)* as follows:

$$InverseCost(t_i, S) = Cost(S) - \sum_{j=1}^{i} Cost(t_j) \quad (10)$$

A. Measure for test reduction

For each test case t_i in the test suite S, the measures *cumCCov(t_i)*, *cumRCov(t_i)*, and *InverseCost(t_i)* are computed considering the position of t_i in S. Then, for each measure above, we computed the area of the curves obtained by plotting in a *Cartesian* plan the values of the metric (on X axes) with respect to the test cases in *suite*$_S$ (Y axes). To get a numerical approximation of that area, we used the *Trapezoidal* rule [8]. It computes the area of a curve as the area of a linear function that approximates that curve.

For a test suite S and each defined cumulative measure, the area (*AUC* in the following) estimates: the code coverage *AUCcumCCov(S)*, the requirements coverage *AUCcumRCov(S)*, and the execution cost *AUCInverseCost(S)*. The area indicates how fast the test suite S converges. The larger *AUC*, the better is.

B. Multi-Objective Reduction

The evaluation of all the possible test case subsets on the three dimensions could be expensive even if in case of non-large test suites. Hence, we propose the use of a multi-objective optimization to prioritize test cases according to the three identified measures. Specifically, we rely on the Non-dominated Sorting Genetic Algorithm II (NSGA-II [9]). Even if different evolutionary algorithm could be used, we resort to NSGA-II since it lets us optimize several, potentially

 ISBN: 978-1-61208-304-9

conflicting, objectives. It has been also widely and successfully used in research work goals similar to ours [10][11].

NSGA-II uses a set of genetic operators (i.e., crossover, mutation, selection) to iteratively evolve an initial population of candidate solutions (i.e., reduced test suites). The evolution is guided by an objective function (called fitness function) that evaluates the quality of each candidate solution along the considered dimensions. In each iteration, the *Pareto* front of the best alternative solutions is generated from the evolved population. The front contains the set of non-dominated solutions, i.e., those solutions that are not inferior to any other solution in *all* considered dimensions. Population evolution is iterated until the maximum number of iterations is reached.

The Pareto front represents the optimal trade-off between the structural, functional, and cost dimensions. The tester can inspect the Pareto front to find the best compromise between having a test case ordering that balance code coverage, requirements coverage, and execution cost or alternatively having a test case ordering that maximizes one/two dimension/s penalizing the remaining one/s. This depends on the testing needs.

Specifically, the technique is set-up as follows:
1. Solution Encoding: A solution is a possible reduced test suite red_S of the application under test. This red_S represents an execution order for a subset of the test cases of the whole test suite *S*. The solution space for the test reduction problem is given by all the permutations of all the possible subsets of the test suite. A reduced test suite is represented as a sequence of integers, where each integer represents a test case identifier and the size of the reduced suite can be set-up by the tester. The maximum number of test cases per suite is a parameter of the algorithm that the tester can customize (e.g., 30% of the whole test suite).
2. Initialization: We randomly initialize the starting population by selecting subsets of test cases among all the possible of test case subsets.
3. Genetic Operators: NSGA-II resorts to three genetic operators for the evolution of the population: mutation, crossover, and selection. The standard operators typically applied for subset of (permutation-based) encoding of solutions are used. As mutation operator, we used the bit-flip mutation: one randomly chosen element of the solution is changed. The adopted crossover operator is the one-point crossover, in which a pair of solutions is recombined by cutting the two solution representations randomly chosen (intermediate) point and swapping the tails of the two cut solutions. We used binary tournament as the selection operator: two solutions are randomly chosen and the fitter of the two is the one that survives in the next population.
4. Fitness Functions: The objective is to maximize the three considered dimensions. Then, each candidate solution in the population (each reduced test suite) is evaluated by our objective function based on: *AUCcumCCov(*red_S*)*, *AUCcumRCov(*red_S*)*, and *AUCInverseCost(*red_S*)*. The larger these values, the faster a reduced test suite converges.

V. EXPERIMENT

To assess the validity of both the technique and the prototype, we conducted an experiment in which we compared test suites reduced with MORE against: *(i)* whole test suites (Full); *(ii)* test suites reduced according to their capability of covering the code (CC) of the target application: CC reduces a test suite S by prioritizing its test cases applying additional code coverage (additional code coverage evaluates each test case of a suite according the code portion that is uniquely covered by it [3]) and then selecting the top-ranked test cases to be part of the reduced suite [3]; and *(iii)* test suites reduced randomly (RA) [12].

A. Experimental Objects

In the study, we used three Java applications AveCalc, LaTazza and iTrust. All applications are distributed online and have been already used in the literature for different purposes [13]. AveCalc manages electronic record books for students: it has 8 classes for 1827 LOCs (excluding comments); it is distributed with 10 textual users' requirements, and 47 JUnit test cases. Latazza is a coffee maker management application: it has 18 classes for 1121 LOCs (excluding comments); it is distributed with 10 textual users' requirements, and 33 JUnit test cases. iTrust Medical Care is a medical application: it has 232 classes for 15495 LOCs (excluding comments); it is distributed with 15 textual users' requirements and with 919 JUnit test cases.

B. Procedure

For each experimental object, we applied the following experimental procedure:
1. Collecting the artifacts: requirements specifications, source code, and test cases.
2. Recovering the traceability links among such software artifacts. As mentioned before, we used the following set-up for LSI: *k*=300; *constant threshold*=0.1.
3. Applying the test reduction techniques (i.e., RA, CC and MORE) to get subsets of the whole test suite, i.e., Full. To balance the number of test cases in the reduced suites, we fixed the size of the reduced test suites (e.g., 30% of Full). Note that we ran MORE with the following set-up: *population size*=2*"test suite size"; *crossover probability*=0.9; *mutation probability*=1/"test suite size". We executed different runs of MORE considering different iterations, that is from *max iterations*=1k to *max iterations*=100k. We, moreover, executed both MORE and RA several times (4 and 20 times, respectively) and evaluated all solutions generated by them. This lets us analyze the average behavior of the techniques (reporting descriptive statistics about the obtained values). MORE has been also executed by weighting the requirements coverage (i.e., using *WRCovD* as the measure for the requirements coverage) according to a requirements prioritization defined by one tester not involved in the rest of the study.
4. Injecting faults in the source code of the application. We injected 15, 15 and 21 faults in AveCalc, LaTazza, and iTrust, respectively. This task was accomplished by an author not involved in the rest of the study. Further details are not provided for space reason (see also [18]).
5. Executing all the test suites in the faulty applications and collecting information about the different evaluation criteria.
6. Repeating the experiment considering several size of the reduced suites: 10%, 20%, 30% and 40% of Full and also after having perturbed the traceability links recovered by MORE (i.e., robustness evaluation).

 ISBN: 978-1-61208-304-9

C. Measures Used for the Comparison

The comparison has been performed with respect to the following evaluation criteria and metrics:

- **Size** ($Size(S)$): *What is the size of the reduced suites?* $Size(S)$ estimates the test effort required to execute the suite S. It is computed as the number of test cases of S.

- **Effectiveness** ($Effect(S)$): *What is the capability of the reduced suites in discovering (injected) faults?* $Effect(S)$ measures the capability of S to reveal (injected) faults. It is evaluated by considering two metrics: $Fault(S)$, the number of revealed faults; and $rFDC(S)$, the fault detection capability rate of S. $rFDC(S)$ is computed as follows:

$$rFDC(S) = \frac{\sum_{f \in F} \frac{FR^f(S)}{|S|}}{|F|} \quad (11)$$

$FR^f(S)$ is the set of test cases in S that reveals the fault f. F is the set of all known faults. $rFDC(S)$ gives us an idea about the capability in revealing faults of the test cases of the suite S. The higher the value of both $Fault(S)$ and $rFDC(S)$, the greater the capability to find faults of the suite S is, that indicates a highly effective suite.

- **Sensitivity** ($Sens(S)$): *What is the capability of the reduced suites of discovering faults affecting top-relevant application requirements?* $Sens(S)$ provides an indication of the capability of S in revealing faults having a high severity and relevance with respect to the application requirements, as well as the application business. $Sens(S)$ is evaluated by means of $Fault'(S)$ applied to the subset of the injected faults that affect relevant application requirements.

- **Efficiency** ($Effic(S)$): *What is the efficiency of the reduced suites in discovering faults?* $Effic(S)$ estimates the capability of S in early detecting the faults and it is measured as:

$$Effic(S) = \frac{Fault(S)}{ECost(S)} \quad (12)$$

$Effic(S)$ is the efficiency computes as the number of detected faults $Fault(S)$ divided the time spent to do it $ECost(S)$ (i.e., the time to run the test cases of the suite S). The larger the value, the more efficient the approach is.

- **Artifact coverage**: *What is the capability of the reduced suites of covering the applications artifacts?* It gives an idea about how the test suite covers both the application code ($Code\_Cov(S)$) and requirements ($Reqs\_Cov(S)$). In detail, we measure two metrics: $Code\_Cov(S)$ is measured in terms of executed code statements exercised at least once by the test cases of the suite while $Reqs\_Cov(S)$ is measured in terms of number of requirement exercised at least once by the test cases of the suite.

- **Diversity** ($Div(S1, S2)$): *How differ the reduced suites are?* $Div(S1, S2)$ estimates the difference of the test cases composing the reduced suites $S1$ and $S2$. It is measured by the Levenshtein edit distance [14] (Ld). This distance indicates the minimum number of operations (insert, delete, and replace) to transform a source string into a target string both built using the same alphabet (i.e., representing test cases of suites $S1$ and $S2$ reduced from S). The values of Ld range from 0 (the two strings are the same) to the maximum length of the two strings (the strings are completely different). Given two testing subsets ($Red1_S$ and $Red2_S$) for a suite S with a fixed number n of test cases, Div is computed as:

$$Div(Red1_S, Red2_S) = (\frac{Ld(Red1_S, Red1_S)}{n}) * 100 \quad (13)$$

- **Robustness** ($Robu(S)$): *How "noise" in the recovered traceability links impacts on the capability of the suites reduced by MORE in revealing faults?* $Robu(S)$ measures the capability of the test reduction technique to adequately work in presence of incomplete or spurious/wrong traceability links (i.e., "noise"). It is evaluated by randomly perturbing the traceability links identified by MORE and re-computing the evaluation criteria for the obtained suites (e.g., effectiveness, efficiency).

- **Settings**: *How the MORE parameter settings can influence the obtained suites in revealing faults?* It gives an indication about how to set-up MORE to make it effective and efficient in revealing faults. With the aim of studying how MORE works in different settings we considered, in particular, different number of iterations of the evolutionary algorithm implemented by MORE and different size of the test suites reduced.

D. Results

Table I summarizes the achieved results in terms of: minimal, median, and maximal values for some of the collected measures (e.g., effectiveness, sensitivity) for the three applications. On the other hand, Figure 1 plots the number of faults revealed by each technique for the three applications. Notice that these results are for the reduced suites containing 30% of Full suites. However, similar results and plots have been collected also for reduced suites having different size, i.e., 10%, 20%, 40% of Full suites. Figure 2 shows the distribution of code coverage and discovered faults for AveCalc at increasing size of the reduced suite (i.e., from 10% to 40% of the Full suite); similar plots have been obtained for all considered metrics and applications. Finally, Figure 3 shows the distributions of discovered faults and efficiency for AveCalc by considering: (i) the reduced suite that is constituted by 30% of the Full size; and (ii) different iterations of our test reduction algorithm: 1k, 4k, 10k and 100k. Similar plots have been obtained for all metrics and applications.

- **Effectiveness**. Table I (values in bold) and the corresponding plots for AveCalc, LaTazza and iTrust in Figure 1 show that the suites reduced with MORE overcome, in most of the cases, the ones reduced by CC and RA while, in few cases, its result is comparable with the best suites obtained from CC and RA. The results achieved by CC and RA are generally worse. We observe that the capability in revealing faults of suites reduced with MORE (and using 30% of the Full test cases) is, at least, double with respect to the other reduced suites, considering the minimal number of revealing bugs per suite. In particular, the suites reduced with RA have an highly variable capability of revealing faults, with respect to those achieved by MORE. This suggests also that MORE can improve the capability of test suites reduced by CC and RA in revealing faults by explicitly optimizing them with respect to code and requirements coverage and execution time as well. However, the good results achieved in few cases by RA, in terms of revealed faults, indicates that improvements are still possible.

 ISBN: 978-1-61208-304-9

TABLE I. SUMMARY OF THE ACHIEVED RESULTS FOR THE REDUCED SUITES HAVING SIZE 30% OF THE SUITES: FULL

	AveCalc				LaTazza				iTrust			
	Full	RA	CC	MORE	Full	RA	CC	MORE	Full	RA	CC	MORE
Size (%)												
	100	30			100	30			100	30		
Effectiveness												
$Fault_{min}$	15	1	6	6	15	2	6	5	21	1	7	3
$Fault_{med}$	-	5	-	8	-	4	-	6	-	4	-	6
$Fault_{max}$	-	8	-	**10**	-	**8**	-	**8**	-	**11**	-	10
$rFDC_{min}$	0.053	0.009	0.05	0.05	0.044	0.01	0.05	0.04	0.0019	0.0002	0.0019	0.0007
$rFDC_{med}$	-	0.004	-	0.08	-	0.04	-	0.05	-	0.0014	-	0.0014
$rFDC_{max}$	-	0.08	-	**0.09**	-	0.07	-	**0.07**	-	**0.0026**	-	0.0024
Sensitivity												
$Fault'_{min}$	6	0	4	2	6	0	3	2	12	1	7	2
$Fault'_{med}$	-	3	-	3	-	2	-	3	-	5	-	5
$Fault'_{max}$	-	**4**	-	**4**	-	**4**	-	**4**	-	7	-	**10**
Efficiency												
$Effic_{min}$	0.83	0.8	0.95	1.5	0.62	0.8	2.2	1.9	0.075	0.011	0.076	0.041
$Effic_{med}$	-	1.1	-	2	-	1.5	-	2.4	-	0.059	-	0.083
$Effic_{max}$	-	1.7	-	**2.5**	-	2.6	-	**3.2**	-	0.132	-	**0.133**
Artifact Coverage												
$Code\_Cov_{min}$	426	414	**426**	419	316	230	301	233	7772	4602	**7430**	4690
$Code\_Cov_{med}$	-	420	-	424	-	284	-	296.5	-	4998.5	-	5681
$Code\_Cov_{max}$	-	**426**	-	**426**	-	308	-	**312**	-	5422	-	6095
$Reqs\_Cov_{min}$	7	6	**7**	6	5	3	**5**	4	**14**	**14**	**14**	**14**
$Reqs\_Cov_{med}$	-	**7**	-	**7**	-	**5**	-	**5**	-	**14**	-	**14**
$Reqs\_Cov_{max}$	-	**7**	-	**7**	-	**5**	-	**5**	-	**14**	-	**14**
Robustness												
$Fault_{min}$	15	-	-	6	15	-	-	6	21	-	-	2
$Fault_{med}$	-	-	-	7	-	-	-	7	-	-	-	4
$Fault_{max}$	-	-	-	8	-	-	-	8	-	-	-	7

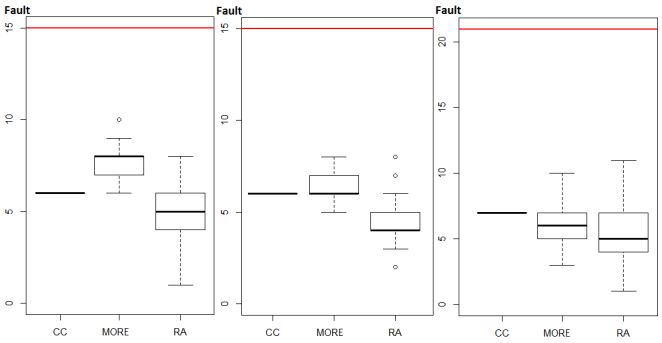

Fig. 1. Boxplots of Faults for AveCalc (left), LaTazza (center) and iTrust (right). The solid line indicates the result of Full.

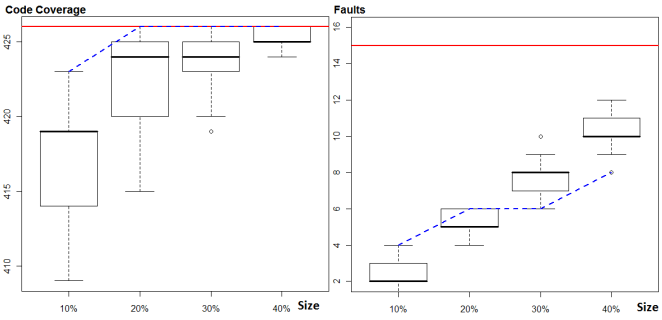

Fig. 2. AveCalc: results at increasing suite size. The solid line indicates the result of Full, the dashed one the result of CC.

- **Sensitivity**. Table I shows that the suites reduced with MORE overcome the ones reduced by CC and RA in terms of minimal number of severe faults impacting top-three relevant requirements (identified by one tester not involved in the rest of the study), and for iTrust also in terms of maximum number of revealed faults.

- **Efficiency**. Table I shows that the suites reduced with MORE always overcome all the other suites (reduced and full ones) in terms of efficiency in revealing faults, i.e., they have required less time to reveal each fault.

- **Artifact coverage**. Table I shows that the suites reduced with

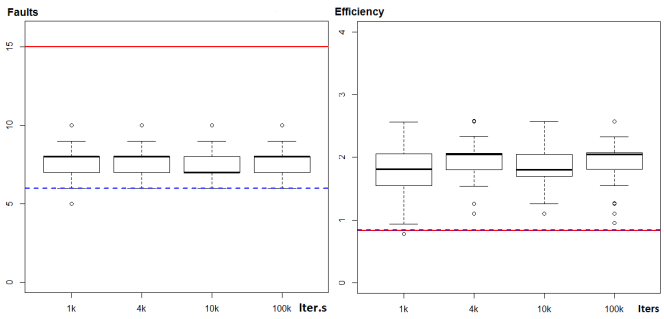

Fig. 3. AveCalc: results at increasing iterations. The solid line indicates the result of Full, the dashed one the result of CC.

MORE achieved a good coverage degree of the application artifacts, i.e., code and requirements. In particular, we can observe that in the considered applications, reduced suites composed of 30% of test cases of the Full suites have the capability to cover: (i) almost all the application requirements used in the study (i.e., more than 60% of requirements); and (ii) a relevant portion of the application source code (i.e., more than 59% of requirements). By manually inspecting test suites and application requirements, we observed that the suites contain redundant test cases, that is test cases that exercise the same portion of code but using different input values and oracles. In addition, some of the used textual application requirements represent quite high-level descriptions of requirements and they do not present too many details, thus they shown high similarity with several test cases, according to LSI.

- **Diversity**. Table II shows the values collected for Div. The test suites reduced by MORE seems to be highly different from the ones reduced with the other techniques. In particular, the high value of the minimal diversity (i.e., 42%), achieved in all the applications by the suites reduced with MORE and CC, suggests a substantial difference of the composition of the test suites reduced by MORE with respect to the ones generated by the single-objective (i.e., CC) technique. While, the high value of the minimal diversity (i.e., 85%), achieved in all the applications by the suites reduced with MORE and RA, suggests that some of the suites reduced by RA are strongly

 ISBN: 978-1-61208-304-9

TABLE II. AVERAGE RESULTS ABOUT *DIV*

DIV	AveCalc	LaTazza	iTrust
MORE - RA	92.4 ÷ 100	85 ÷ 100	99.1 ÷ 100
MORE - CC	42 ÷ 100	42.7 ÷ 100	98.2 ÷ 100

similar to the ones generated by MORE.

- **Robustness:**. Table I shows that the suites reduced with MORE revealed less faults, on average, than the corresponding suites reduced using the actual traceability links recovered by MORE. However, for LaTazza the number of revealed faults increases of few points, this indicates the existence of traceability links incorrectly recovered. Further experimentation needs to be devoted to evaluate and detect such links.

- **Settings:**. Figure 2 shows, as example, the results of the suites reduced with MORE for AveCalc at different suite size, respectively for the code coverage measure (left figure) and for the discovered faults (right figure). Similar plots have been computed for all evaluation criteria and applications. From these plots, we observe that the suites reduced with MORE at 20,30% of Full suite size achived results almost comparable to the same Full suites, in terms of artifacts coverage, and reasonably high results in terms of effectiveness and efficiency. Conversely, the MORE suites built using less than 20% of Full performed better, in terms of revealed faults, than CC. We argue that this is mainly due to the fact that the suites reduced with MORE by considering, e.g., 10% of Full size have a quite limited coverage of the application artifacts, than CC (Figure 2-left the plot of code covered by MORE and CC). Furthermore about the technique settings, Figure 3 shows that increasing the maximum number of iterations of the evolutionary algorithm implemented by MORE does not allow achieving better results in term of discovered faults and suite efficiency (see the plots of all the three applications).

- **Final remarks**. In conclusion, the results achieved in the experiment show that: *(i)* consistently with the existing literature [15], the multi-objective optimization is overall effective in reducing test suites by balancing different dimensions and *(ii)* MORE achieves good results and it tends to outperform CC and RA, even when a non-trivial suite reduction (e.g., 20/30% of the full suite) is considered.

E. Threats to Validity

A possible threat that might affect the validity of the achieved results is represented by the injection of faults in the application code and their distribution. Different sets of faults can potentially lead to different results. To reduce this threat, one of the authors (not involved in the rest of the study) injected faults in the application code. An other issue could be also represented by the non-deterministic behavior of the reduction techniques used. To reduce these biases, we applied MORE and RA several times and then evaluated all the generated solutions to study the average trend. Finally, both the size and complexity of the considered applications may threaten the validity and the generalization of our results.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a multi-objective technique to reduce test suites. The technique reduces test suite considering the coverage of source code and application requirements, and the cost to execute test cases. An IR-based traceability recovery approach has been defined and applied to link software artifacts (i.e., requirements specifications, source code, and test cases). A reduced test suite is then determined by using a multi-objective optimization, implemented in terms of NSGA-II. Our technique has been evaluated using Java applications and results are promising. Future work is, however, needed to further assess MORE on bigger software applications and compare our solution with additional test reduction techniques.

REFERENCES

[1] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” Software Testing, Verification and Reliability, vol. 22, no. 2, 2010, pp. 67–120.

[2] J. Karlsson and K. Ryan, “A cost-value approach for prioritizing requirements,” IEEE Software, vol. 14, 1997, pp. 67–74.

[3] D. Jeffrey and N. Gupta, “Improving fault detection capability by selectively retaining test cases during test suite reduction,” IEEE Trans. Softw. Eng., vol. 33, no. 2, Feb. 2007, pp. 108–123.

[4] S. McMaster and A. M. Memon, “Call stack coverage for test suite reduction,” in Procs. of Intern. Conf. on Software Maintenance, IEEE Computer Society, 2005, pp. 539–548.

[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, “Indexing by latent semantic analysis,” Journal of the American Society of Information Science, vol. 41, no. 6, 1990, pp. 391–407.

[6] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, “Recovering traceability links in software artifact management systems using information retrieval methods,” ACM Trans. Softw. Eng. Methodol., vol. 16, no. 4, 2007.

[7] I. Aaqib, K. Farhan M., and K. Shahbaz A., “A critical analysis of techniques for requirement prioritization and open research issues,” International Journal of Reviews in Computing, vol. 2, no. 1, 2009, pp. 8 – 18.

[8] K. Atkinson, An Introduction to Numerical Analysis, 2nd ed, Wiley, 1989.

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” IEEE Trans. Softw. Eng., vol. 6, no. 2, 2002, pp. 182–197.

[10] A. Marchetto, C. Di Francescomarino, and P. Tonella, “Optimizing the trade-off between complexity and conformance in process reduction,” in Procs. of Intern. Conf. on Search based software engineering, Berlin, Heidelberg: Springer-Verlag, 2011, pp. 158–172.

[11] Y. Zhang and M. Harman, “Search Based Optimization of Requirements Interaction Management,” in In Procs. Intern. Symposium on Search Based Software Engineering, IEEE Computer Society, 2010, pp. 47–56.

[12] A. Arcuri, M. Iqbal, and L. Briand, “Random testing: Theoretical results and practical implications,” IEEE Trans. Softw. Eng., vol. 38, no. 2, 2012, pp. 258 –277.

[13] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, M. Ceccato, and C. A. Visaggio, “Are fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks,” in Procs. of Intern. Conf. on Software Engineering, ACM, 2008, pp. 361–370.

[14] V. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” Soviet Physics Doklady, vol. 10, 1996, p. 707.

[15] S. Yoo and M. Harman, “Pareto efficient multi-objective test case selection,” in Procs. of Intern. Symposium on Software testing and analysis, ACM, 2007, pp. 140–150.

[16] L. de Souza, P. de Miranda, R. Prudencio, and F. de Barros, “A multi-objective particle swarm optimization for test case selection based on functional requirements coverage and execution effort,” in Procs. of Intern. Con.e on Tools with Artificial Intelligence, 2011, pp. 245 –252.

[17] X. MA, Z. He, B. kui Sheng, and C. Ye, “A genetic algorithm for test-suite reduction,” in Procs. of Inter. Conf. on Systems, Man and Cybernetics, vol. 1, 2005, pp. 133–139.

[18] M. M. Islam, A. Marchetto, A. Susi, and G. Scanniello, “A multi-objective technique to prioritize test cases based on latent semantic indexing,” in Procs. of European Conference on Software Maintenance and Reengineering, IEEE Computer Society, March 2012, pp. 21 –30.

 ISBN: 978-1-61208-304-9