RN/13/04
March 4, 2013

Research
Note

Cost-cognisant Test Suite Reduction for Service-centric Systems

Mustafa Bozkurt

Telephone: +44 (0)20 7679 0332
Fax: +44 (0)171 387 1397
Electronic Mail: m.bozkurt@cs.ucl.ac.uk
URL: http://www.cs.ucl.ac.uk/staff/M.Bozkurt/,

Research Note
RN/13/04

Cost-cognisant Test Suite Reduction for Service-centric Systems

04/03/2013

Mustafa Bozkurt

Abstract

This paper concerns with runtime testing cost due to service invocations, which is identified as one of the main limitations in Service-centric System Testing (ScST). Unfortunately, most of the existing work cannot achieve cost reduction at runtime as they are aimed at offline testing. The paper introduces a novel cost-aware pareto optimal test suite minimisation approach for ScST aimed at reducing runtime testing cost. The approach adapts traditional multi-objective minimisation approaches to ScST domain by formulating ScST concerns, such as invocation cost and test case reliability. The paper presents the results of an empirical study to provide evidence to support two claims: 1) the proposed solution can reduce runtime testing cost, 2) the selected multi-objective algorithm HNSGA-II can outperform NSGA-II. In experimental analysis, the approach achieved reductions between 69% and 98.6% in monetary cost of service invocations during testing. The results also provided evidence for the fact that HNSGA-II can match the performance or outperform both the Greedy algorithm and NSGA-II in different scenarios.

Abstract

This paper concerns with runtime testing cost due to service invocations, which is identified as one of the main limitations in Service-centric System Testing (ScST). Unfortunately, most of the existing work cannot achieve cost reduction at runtime as they are aimed at offline testing. The paper introduces a novel cost-aware pareto optimal test suite minimisation approach for ScST aimed at reducing runtime testing cost. The approach adapts traditional multi-objective minimisation approaches to ScST domain by formulating ScST concerns, such as invocation cost and test case reliability.The paper presents the results of an empirical study to provide evidence to support two claims: 1) the proposed solution can reduce runtime testing cost, 2) the selected multi-objective algorithm HNSGA-II can outperform NSGA-II. In experimental analysis, the approach achieved reductions between 69% and 98.6% in monetary cost of service invocations during testing. The results also provided evidence for the fact that HNSGA-II can match the performance or outperform both the Greedy algorithm and NSGA-II in different scenarios.

Keywords

web service testing, test suite minimisation, realistic test data

Department of Computer Science
University College London
Gower Street
London WC1E 6BT, UK

1 Introduction

According to literature, one of the limitations of ScST is the cost associated with invoking services [3]. Increased test frequency for ScS exacerbates the severity of the issues regarding testing cost. The cost of invoking services during testing is a major problem at composition level. Solutions aimed at reducing the cost of testing, such as simulated testing have previously been proposed [8, 9, 10, 11, 12, 15]. However, these approaches do not eliminate the need for runtime testing (testing with real services).

One widely studied solution aimed at reducing runtime testing cost is test suite minimisation [24]. The purpose of test suite minimisation and prioritisation is to reduce testing cost through removing redundant test cases. In test suite minimisation, there are concerns such as retaining coverage, test suite effectiveness or fault detection capability, execution time and resource usage, such as memory. According to the literature, problems where there are multiple competing and conflicting concerns can be investigated using pareto optimal optimisation approaches [7].

A tester is often faced with making decisions based on multiple test criteria during runtime testing. This is due to the fact that, in many situations, it is not expected that the tester aims to achieve a single goal during the testing process. According to the literature, one goal that is often used is code coverage. Achieving a high coverage is often regarded as a major goal in testing [21, 17]. However, achieving a high coverage is expensive and expecting to achieve 100% coverage might not be realistic when testing complex systems.

The cost of a test suite is one of the most important criteria, since a tester wants to get maximum value (e.g. number of branches covered) from the execution of a test suite [7]. In the literature, the cost of a test suite is often associated with the time it takes to execute it [24]. However, with the introduction of web services, service compositions and online APIs, the concept of monetary cost of testing is starting to gain acceptance in software testing literature.

The reliability (also referred to as quality) of a test suite is also an important concern. Previously, we introduced a formulation for the reliability score of a test input based on the reliability of its sources [2]. We believe reliability is an important aspect especially in ScST, because it can help reduce the testing cost. A test suite with high reliability might reduce the cost of testing in two ways:

1. Human-oracle reduction due to the tester's trust in the test input source. If the test data used in testing has a high reliability, the tester will not need to inspect all test inputs for errors such as invalidity and conformance to input requirements (e.g. input format).
2. Avoiding extra testing cost due to erroneous test inputs. The extra cost in this case occurs due to service invocations with invalid test data during the execution and the tester's investigation of the unexpected test results that might occur as a result of invalid inputs.

For instance, consider the case of U.S. ZIP codes. Zip codes can be decommissioned or new ones might be assigned to new areas over time. As a result, during the testing of an ScS that requires ZIP codes as inputs, only valid codes must be used. In this scenario, if the ZIP codes (used in testing) are generated from unreliable services, then the test suite's reliability will be low, which means the test suite might include decommissioned or invalid ZIP codes. It is safe to assume that testing with decommissioned ZIP codes can be classified as unnecessary execution (except if the tester is performing robustness testing), which is likely to produce false positives. The results from our experimental studies for ZIP codes (discussed in Section 4) provided some evidence for the correctness of this assumption. The problem with false positives is that they might cause the tester to invest time in investigating the reasons for unexpected output(s).

In order to avoid the side effects of extensive runtime testing, a test suite minimisation approach could be used to remove 'redundant' test cases; those which merely cover previously covered features. On the other hand, to avoid side effects of a low reliability test suite, the selected approach must be able to manage multiple objectives. Thus, we propose to adapt the multi-objective test suite minimisation approach of Yoo

and Harman [24] to ScST with the objectives, such as cost of service invocation, branch coverage and test suite reliability.

In this paper, we introduce a pareto-optimal, multi-objective test suite minimisation approach to ScST aiming at reducing the runtime testing cost. The advantages of the proposed application of multi-objective test suite minimisation for ScS are:

1. Reduced cost in runtime testing through test suite minimisation.
2. Its ability to discover trade-offs between cost of test runs and system coverage.
3. Its ability to select a more reliable test suite without increasing the cost and affecting the coverage of the test suite.

The rest of this paper is organised as follows. Section 2 briefly introduces the concept test suite minimisation and explains the proposed multi-objective test suite minimisation approach for ScST. Section 3 presents our case studies, research questions and our method of investigation. Section 4 presents the results from our experiments and answers the research questions. Section 5 concludes the paper.

2 Multi-Objective Test Suite Minimisation for Service-centric Systems

In this section, we explain the concept of test suite minimisation, our approach and present our objective functions.

2.1 Test Suite Minimisation and HNSGA-II

Test suite minimisation (or test suite reduction) techniques aim to reduce the size of a test suite by eliminating redundant test cases [24]. Test suite minimisation is considered as a hitting set (or set cover) problem which is an NP-complete problem and defined as follows:

Input: A test suite T =$\{t_1, ..., t_n\}$ and a set of testing requirements R = $\{r_1, ..., r_n\}$ which need to be satisfied in order to provide the desired level of testing.

Goal: To find a representative set of test cases T_0, from T ($T_0 \subseteq T$) that satisfies all requirements. In order to satisfy all requirements, each r_i must at least be satisfied by one of the test cases that belongs to T_0. The effect of minimisation is maximised when T_0 is the minimal hitting set of the test cases in T.

Due to the test suite reduction problem (which is a minimal set cover problem) being NP-complete, the use of heuristics is proposed by many researchers [24]. According to the literature, another well known solution to the set cover problem is the greedy approximation. Yoo and Harman [22] proposed a solution (using an algorithm called 'HSNGA-II') which combines these two solutions into a pareto-optimal test suite minimisation approach. HNSGA-II uses the additional greedy algorithm, described in Algorithm 1 along with multi-objective algorithms.

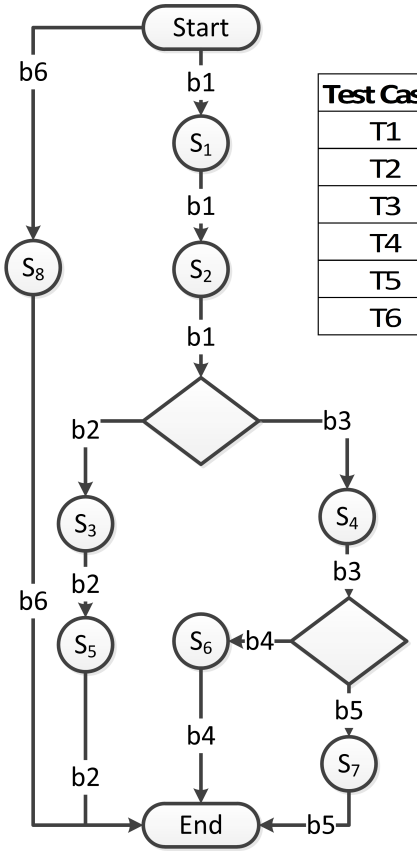
Algorithm 1 Additional Greedy Algorithm $(\mathcal{U}, \mathcal{S})$ [23]

1: $C \leftarrow \emptyset$ ▷ covered elements in $\mathcal{U}$
2: **repeat**
3: $j \leftarrow min_k(cost_k/|S_k - C|)$
4: add S_j to solution
5: $C = C \bigcup S_j$
6: **until** $C = \mathcal{U}$

Where $\mathcal{U}$ is the universe, $\mathcal{S}$ is the set that contains $S_1, S_2, ..., Sn$ (with execution costs $cost_1, cost_2, ..., cost_n$) which cover subsets of $\mathcal{U}$, such that $\bigcup_i S_i = \mathcal{U}$. The assumption in this scenario is the existence of a subset

of $\mathcal{S}$ which covers all elements of $\mathcal{U}$. The additional algorithm is cost cognisant, thus it does not just pick the subset that covers the most elements, but it aims at finding the subset that provides maximum coverage increase with the lowest cost at each iteration (at Line (4)) [23].

HNSGA-II is a variant of the standard NSGA-II algorithm and it may be more effective in multi-objective minimisation problems compared to NSGA-II. HNSGA-II combines the effectiveness of greedy algorithm for set cover problems with NSGA-II's global search. Results from the execution of additional greedy algorithm are added to the initial population of NSGA-II in order to create an initial population with better solutions compared to a 'random only' population. The goal in using a better initial population is to guide NSGA-II to a better approximation to the optimal pareto front. This process can be especially beneficial in problems with very large search spaces.



Test Case	Reliability	Branches	Services	Cost
T1	0.90	b1, b2	S1,S2,S3,S5	$12.10
T2	0.87	b6	S8	$2.04
T3	0.91	b1,b3,b5	S1,S2,S4,S7	$17.15
T4	0.76	b1,b3,b4	S1,S2,S4,S6	$11.35
T5	0.99	b6	S8	$2.04
T6	0.93	b1, b2	S1,S2,S3,S5	$12.10

Figure 1: Example test suite reduction scenario for a ScS. The table depicts test cases in the suite with their reliability, branch coverage and execution cost calculated. For the given test suite (T1,...,T6) it is expected that test cases T1 and T2 will be eliminated to get the optimal test suite (T3,T4,T5,T6) which achieves 100% coverage with lowest cost and highest reliability.

2.2 Proposed Approach

Our approach consists of two stages: test suite artefact calculation and multi-objective minimisation. After test suite generation, our approach requires the calculation of three measurements in order to apply multi-objective approaches. These are coverage, reliability and execution cost of each test case.

The reliability score of a test case is based on the reliability of its inputs. The reliability of each input is calculated using the formulation we described in our previous work [1, 2]. Previously, we introduced the concept of service-centric test data generation addressing the issues of automation in realistic test data generation [1]. The proposed solution called ATAM composes existing services to generate the required realistic test data. ATAM also calculates the reliability of each generated realistic input based on the

services used in the input generation process. Details of our formulation for the input reliability score is available in an earlier paper [2].

Unlike reliability, execution cost and branch coverage cannot be acquired from an external source. The easiest way of acquiring this information is by executing the whole test suite. Unfortunately, performing a runtime execution for the whole test suite in order to measure its artefacts will increase the overall cost of testing which is an unacceptable side effect for an approach that aims to reduce testing cost. In order to avoid this cost, we propose the use of simulated testing using mock/stub services. Using stub/mock services will allow us to measure branch coverage and service invocation information for each test case without incurring additional costs.

Service invocation costs can occur in several ways (based on the type of contract agreed between the provider and the integrator). Two of the most prominent payment plans used at present are: pay per-use and invocation quota-based plans. As a result, two different cost calculation functions are introduced. Details of these two payment plans and the cost calculation associated with them are discussed in Section 2.3.

After completing stub/mock service generation, a simulated run for the test suite is performed in order to measure testing artefacts for each test case. These measured values then used in the optimisation process to determine the pareto optimal sets of test cases. An illustration of an example ScS and its test suite with the test case measurement is depicted in Figure 1.

In order to adapt Yoo and Harman's minimisation approach to ScST, we modified the original greedy algorithm by replacing its objective functions with the objective functions from our approach. We used the following algorithms for the 2-objective and the 3-objective optimisation scenarios. The additional algorithm for 2-objective optimisation (described in Algorithm 2) uses the cost and coverage calculation algorithms discussed in Section 2.3.

Algorithm 2 2-objective Additional Greedy Algorithm

Require: Test suite S
1: DEFINE current fitness
2: DEFINE test case subset $\mathcal{S}' := \emptyset$
3: **while** not stopping rule **do**
4: current fitness := 0
5: **for all** test cases in the test suite $\mathcal{S}$ **do**
6: **if** test case T is not in $\mathcal{S}'$ **then**
7: current fitness := coverage score of $\mathcal{S}'$ ($Cov_{\mathcal{S}'}$)
8: ADD T to $\mathcal{S}'$
9: new fitness := $\frac{Cov_{\mathcal{S}'} - \text{current fitness}}{\text{cost of } \mathcal{S}'}$
10: **end if**
11: **if** new fitness is better than current fitness **then**
12: current fitness := new fitness
13: MARK T as selected
14: REMOVE T from $\mathcal{S}'$
15: **end if**
16: **if** No test case is selected **then**
17: End
18: **else**
19: ADD T to $\mathcal{S}'$
20: **end if**
21: **end for**
22: **end while**

The 3-objective additional algorithm (described in Algorithm 3) considers an additional objective, reliability. In this algorithm, the objectives coverage, cost and reliability are combined into a single objective using the weighted-sum model. In our experiments, both coverage and reliability objectives were given equal weights.

Algorithm 3 3-objective Additional Greedy Algorithm

Require: Test suite S
DEFINE current fitness
DEFINE test case subset $\mathcal{S}' := \emptyset$
while not stopping rule **do**
    current fitness := 0
    for all test cases in the test suite $\mathcal{S}$ **do**
        if test case T is not in $\mathcal{S}'$ **then**
            coverage fitness := coverage score of $\mathcal{S}'$ ($Cov_{\mathcal{S}'}$)
            reliability fitness := reliability score of $\mathcal{S}'$ ($Rel_{\mathcal{S}'}$)
            ADD T to $\mathcal{S}'$
            coverage fitness := $\frac{Cov_{\mathcal{S}'} - \text{coverage fitness}}{\text{cost of } \mathcal{S}'}$
            reliability fitness := $\frac{Rel_{\mathcal{S}'} - \text{reliability fitness}}{\text{cost of } \mathcal{S}'}$
            new fitness := $\frac{\text{coverage fitness} + \text{reliability fitness}}{2}$ ▷ reliability and coverage are given equal weight
        end if
        if new fitness is better than current fitness **then**
            current fitness := new fitness
            MARK T as selected
            REMOVE T from $\mathcal{S}'$
        end if
        if No test case is selected **then**
            End
        else
            ADD T to $\mathcal{S}'$
        end if
    end for
end while

As mentioned, HNSGA-II uses the results from the greedy algorithm runs as an initial population. In the second stage of our approach, we run the greedy algorithm and feed its results to NSGA-II algorithm which produces the pareto optimal front enabling the tester to investigate the trade-offs between the measured testing artefacts. In the cases where the size of resulting set from greedy algorithm is less than the required population size for NSGA-II, we compensate for this shortcoming by adding randomly generated solutions to the greedy results.

2.3 Objective Functions

Branch coverage is calculated as the percentage of branches of ScS under test covered by the given test suite. In our approach, we considered coverage as an objective rather than a constraint, in order to allow the tester to explore all the possible solutions on the pareto-optimal front. The following objective function is used for branch coverage since our aim is to maximise the coverage of the test suite.

$$Maximize \ \frac{\text{branches covered by test suite}}{\text{total number of branches}} \tag{1}$$

The objective function for the cost is not as straightforward as branch coverage because it has multiple

options for service invocation cost. Several different payment plans might exist for service usage. However, we considered only the two prominent ones: pay per-use and quota based payment plans.

Pay per-use plan: In this case, the integrator is charged for each service invocation individually. The total cost of executing a test case is calculated as the total cost of services invoked by the test case and is formulated as:

$$cs(tc_m) = \sum_{i=1}^{n} X_{S_i} * C_{S_i}$$

where n is the number of services invoked by executing the test case tc_m, S_i is the ith executed service, C_{S_i} is the cost of invoking service S_i and X_{S_i} is the number of times service S_i invoked by this test case.

The cost for each service can be determined by discovering available services and their prices. At runtime, it is safe to assume that multiple alternatives for each service in the composition will be discovered. As a result, the tester might not know which services will be invoked at runtime. However, the tester needs to assign a static service cost to each service in the composition in order to calculate the cost artefact for each test case. In order to determine the price of each service, the tester might choose to use one of several criteria, such as using maximum, average or minimum price of the discovered services alternatives. This is flexibility essential because the tester might choose to change the runtime service selection criteria during the test runs (to force the invocation of low-cost services) in order to reduce the cost of testing. In this case, the lowest invocation costs for each service in composition is used to calculate cost of test cases. However, the tester might also choose to use the average or maximum invocation costs if a more realistic runtime testing is desired.

The objective function for pay-per use plan is formulated as:

$$Minimise \ \sum_{i=1}^{k} cs(tc_i) \tag{2}$$

where k is the total number of test cases and $cs(tc_i)$ is the total cost of executing the ith test case in the test suite.

Invocation quota based plan: In this case, the integrator pays a subscription fee for a number of service invocations within a period of time (such as monthly or annually). In our scenario, we presume that all the services used in the composition are selected from a single provider and a total invocation quota applies to all services rather than an individual quota for each service. The objective function for this plan is formulated as:

$$Minimise \ \text{number of services invocations.} \tag{3}$$

Generating test data using ATAM also enables the use of another test case artefact: reliability. Reliability of a test case is based on the reliability of its inputs. Reliability of a test input is calculated by ATAM as the combined reliability of the data sources used in generation of this input. The reliability calculation and data source selection in ATAM are discussed elsewhere [2].

Each test case might include a combination of inputs generated using ATAM and user generated inputs. In the case of user generated inputs we consider the input to be 100% reliable and for ATAM generated inputs the reliability score is provided by ATAM. The reason behind considering the tester input as 100% reliable is our assumption of the tester's likely verification of the input data before test execution. Since we do not modify the tester inputs and use them as they are, we did not foresee any reason for having variations in the reliability score of the tester generated data.

In light of these possible cases, a reliability function covering these two cases is formulated as:

$$rf(in_x) = \begin{cases} 1.0 & \text{if } in_x \text{ is user generated} \\ \text{ATAM score} & \text{if } in_x \text{ is generated using ATAM} \end{cases}$$

where $rf(in_x)$ is the reliability score of the input in_x.

The reliability score of each test case is calculated as the average reliability of its inputs, and is formulated as:

$$rel(tc_m) = \frac{1}{y} \sum_{i=1}^{y} rf(in_i)$$

where y is the number of test inputs and $rf(in_i)$ is the reliability of the ith input (in_i) of the test case tc_m.

Reliability of a test suite is calculated as the average reliability of its test cases. Since our aim is to increase the reliability of the test suite, the objective function for test suite reliability is formulated as:

$$Maximise \ \frac{1}{z} \sum_{i=1}^{z} rel(tc_i) \tag{4}$$

where z is the number of test cases in the test suite and $rel(tc_i)$ is the reliability of the ith test case (tc_i) in the test suite.

2.4 Mutli-Objective Algorithm and Parameters

To implement and evaluate our approach, we used the popular ECJ framework [5] which provides a built-in NSGA-II algorithm. We used a single population with a size of 2000 and set the number of generations to 100. After some tuning, we found that the ideal parameters that provide the most diverse solutions for our problem are: 5% mutation probability for each gene and single-point crossover with 90% crossover probability.

As mentioned, the only difference between HNSGA-II and NSGA-II is the initial population. The initial population of NSGA-II is generated by ECJ's internal mechanism. However, the initial population for HNSGA-II requires the use of additional greedy algorithms. The results from the additional greedy algorithm combined with the randomly generated solutions (in order to match the stated population size) are fed to EJC as the initial population of NSGA-II algorithm.

3 Empirical Studies

In this section, we introduce the case studies we used in our experiments, present the research questions we asked and explain our method of investigation.

3.1 Case Study

In order to evaluate our approach, we selected two case studies with different characteristics. The reason behind this selection is to observe the effectiveness of our approach in different scenarios by providing results that might challenge the findings from the other case study.

The first case study (CS1) is an example code used as a benchmark for applications/approaches that aim to achieve branch coverage called 'Complex'. Complex is a artificial test object with complex branching conditions and loops [20]. However, in its original form, Complex is not an ScS. In order to evaluate our approach, we transformed Complex into an ScS by replacing all mathematical, relational and logical operators with service calls[1]. We choose an existing calculator web service [4] to replace the 4 mathematical

[1]The source code available at http://www0.cs.ucl.ac.uk/staff/M.Bozkurt/files/public/complex_source.zip

operators: addition, subtraction, division and multiplication. For the other five operators, we implemented a web service providing the required services. The list of used services are presented in Table 5.

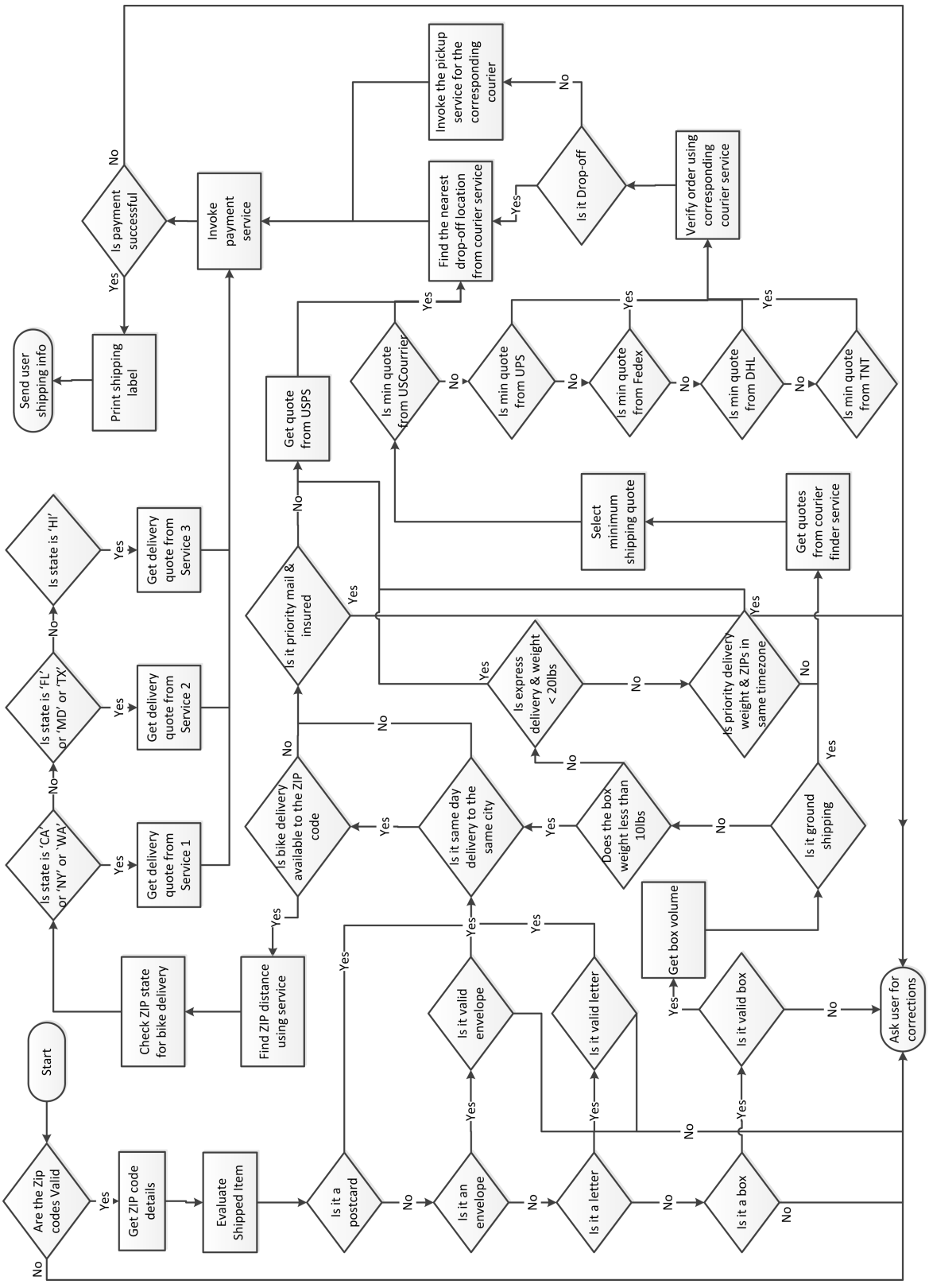


Figure 2: Flowchart of the second case study

The second case study (CS2) is a synthetically created shipping workflow that combines the functionality of a number of available online services [6, 16, 18, 19]. In order to make CS2 as realistic as possible, the

shipping service invokes a combination of existing web services and other synthetically created services. We were required to use synthetic services in order to simulate the functions of existing services that we have restricted access to.

The most important aspect of CS2 is that it works with real-world services, which requires realistic test inputs. The workflow requires a total of 14 inputs and 2 of these inputs are realistic (ZIP codes). The other inputs are user generatable shipping options such as mail type, delivery type, box weight and dimensions. The realism of CS2 is also strengthened by the fact that the ZIP codes used in this study are generated from existing web services. The flow graph for the workflow is depicted in Figure 2 and the 14 web services invoked are presented in Table 6.

As discussed, one of the advantages of ATAM is its ability to generate test data based on the reliability of the test data source. In order to carry out our experiments, we needed to measure the reliability of the services we used for generating test inputs. However, this was not possible for some of the services we used due to access restrictions. In order to overcome this limitation and to increase the realism of our case studies, we need real-world reliability scores. Thus, we measured the reliability of 8 publicly available existing services, presented in Table 1.

Service	**Type**	**Number of errors**	**Reliability Score**
USPS.com	ZIP code validation	0	0.999
Websitemart.com	ZIP code validation	3102	0.744
Zip-codes.com	ZIP code validation	50	0.995
Webservicesx.com	ZIP code validation	727	0.939
NOOA.gov	Weather service	410	0.965
Myweather2.com	Weather service	146	0.987
CYDNE.com	Weather service	1218	0.899
Weatherbug.com	Weather service	550	0.954
Webservicesx.com	State ZIP code info	727	0.939

Table 1: The list of the services used in determining the real-world reliability scores used in our experiments. Services in the list are tested with 12171 ZIP codes belong to 23 U.S. States. The USPS service on the list is accepted as the ground truth for the validity of the ZIP codes. The rest of the ZIP validation services are evaluated with the generated ZIP codes to observe if they correctly identify given ZIP codes' validity. As for the weather services, we observed if they return weather information only for the valid ZIP codes.

There are three main reasons that led us to choose these services for this part of our experiments:

1. The public availability and having no access restrictions, evidently supporting replication.
2. Requiring the same input that can be validated using a service which can be accepted as the ground truth.
3. Similar functionality of services allowing determination of the expected output.

The services in the list are categorised into two main groups based on their functionality; ZIP code verification services and weather services. Verification services check if a given 5-digit US ZIP code is valid. Weather services provide current weather information for a given ZIP code.

Services in the list were tested using 12171 ZIP codes of 23 different U.S. states. The ZIP codes used in reliability analysis are generated using the the 9th service from Table 1. The USPS service in the table is accepted as the ground truth for the validity of the ZIP codes. As a result, the number of errors observed for this service is set to 0 and its reliability score is set to the highest reliability score: 0.999. The rest of the ZIP validation services are evaluated with the generated ZIP codes to observe whether they correctly

identify the given ZIP code's validity. As for the weather services, we only considered the valid ZIP codes in order to maintain consistency and checked if they return weather information for all valid ZIP codes. For the reliability of the last service, we simply counted the number of invalid ZIP codes from the generated outputs.

Input	**Reliability**	
	Positive	**Negative**
A	0.999	0.744
B	0.995	0.939
C	0.965	0.987
D	0.899	0.954
E	1.0	1.0
F	1.0	1.0

Table 2: Reliability values used in the 3-objective evaluation of CS1. Positive and negative values for each input are assumed to be generated by a single web service with the given reliability on the list. The last two inputs are assumed to be human generated thus their reliability scores are 100%.

After acquiring the reliability scores, we generated test inputs for both case studies. As for the test inputs, Complex does not require realistic inputs, but requires 6 integer numbers which can be automatically generated. As a result, we did not use ATAM in generating test inputs for CS1. Instead, we generated the required test inputs using a random number generation method. However, in order to maintain consistency in our experiments, we also evaluated the 3-objective formulation of our approach on CS1. Thus, we needed to generate a fictional scenario where inputs are generated using different services such as the Random.org integer generator [13]. In this scenario, we assumed that the positive and negative values for each of the first four inputs (A to D) are generated using 8 different services. We assigned 8 of the measured reliability scores to these services as presented in Table 2.

For CS2 we determined 3 web service compositions, as presented in Table 3, to generate ZIP codes. The services and test inputs used in evaluating this case study are real-world entities. However, we were unable to measure the actual reliabilities of the three services in the first composition due to access restrictions. As a result, we assigned 3 of the reliability scores from Table 1 to the 3 services in the first composition (in Table 3) and used these scores in the combined reliability score calculation of this composition.

	Service			**Reliability**	
	Publisher	**Input(s)**	**Output**	**Individual**	**Combined**
1	Google Search	Keyword	Search result	0.999	0.881
	Strikeiron	Http address	IP address	0.744	
	FraudLabs	IP address	US location	0.899	
2	Codebump.com	–	US state names	0.954	0.947
	Webservicesx	US state name	ZIP code	0.939	
3	Webservicesx	US area code	ZIP code	0.899	0.899

Table 3: Web service compositions used in generating ZIP codes. Individual reliability scores represent the scores for each service and combined scores represent the reliability of the composition. The combined score of a composition also determines the reliability scores of inputs generated using this composition.

The other needed artefact is the cost of invoking services. This raises the issue of how to choose realistic values for this characteristic. In an earlier publication [2], we discussed this issue of generating realistic invocation cost values for the services used in the experiments and presented a solution where realistic costs values are obtained using costs of existing services as a basis. We adopted the same approach again and used the cost values from Table 4.

Service Group		Price (per query)			
No	**Description**	**Max**	**Company**	**Min**	**Company**
1	Phone verification	$0.300	StrikeIron	Free	WebServiceMart
2	Traffic information	$0.300	MapPoint	Free	MapQuest
3	Geographic data	$0.165	Urban Mapping	$0.010	Urban Mapping
4	Bank info verification	$0.160	Unified	$0.090	Unified
5	IP to location	$0.020	StrikeIron	Free	IP2Location
6	Stock Quote	$0.020	XIgnite	$0.008	CDYNE
7	Financial data	$0.017	Eoddata	$0.007	XIgnite
8	Nutrition data	$0.010	CalorieKing	Free	MyNetDiary
9	Web search	$0.005	Google	Free	Bing

Table 4: Services used as a basis for the synthetically generated case study. The services and the given prices in this table are collected from Remote Methods website [14].

The real-world web service we used in CS1 is a free-to-use web service and provides 4 of the services used in this case study. Unfortunately, the free services presented challenges for maintaining consistency especially on CS1 due to these services being the most invoked services. This problem had an impact on our experiments using a per-use payment plan, causing most of the test cases having the same execution cost even though there were large differences in the number of the services invoked. As a result, we assigned the invocation cost values presented in Table 5 to the services in CS1.

Service		Price
No	**Description**	
1	Multiplication	$0.300
2	Division	$0.300
3	Logical AND	$0.165
4	Logical OR	$0.160
5	Greater Than	$0.020
6	Less Than	$0.020
7	Equal To	$0.017
8	Subtract	$0.010
9	Add	$0.005

Table 5: The invocation costs for the services used in CS1.

For CS2, we used a combination of real service costs (for free to use services) and synthetically generated values (using the same method adopted in CS1) as presented in Table 6.

3.2 Research Questions

We ask the following three questions:

RQ1 Can multi-objective test suite minimisation reduce the testing cost by finding optimal test suite subset(s) for ScST?

RQ2 Is using a test suite with low reliability (containing invalid inputs) likely to generate false positives which might increase the testing cost?

RQ3 Can HNSGA-II algorithm discover dominating solutions compared to NSGA-II and the additional greedy algorithm in our problem?

Service			Price
No	**Name**	**Description**	
1	WebserviceMart	ZIP code verification	Free
2	WebServicesx	ZIP code info	$0.090
3	Bike messenger 1	Find bike messenger	$0.008
4	Bike messenger 2	Find bike messenger	$0.007
5	Bike messenger 3	Find bike messenger	$0.007
6	WebServicesx	ZIP code distance	$0.020
7	USPS	Get delivery price	Free
8	Fedex	Get delivery price	Free
9	TNT	Get delivery price	Free
10	DHL	Get delivery price	Free
11	UPS	Get delivery price	Free
12	Payment system	get card payment	$0.30
13	Label system	print mail label	$0.017
14	Courier finder	find courier	$0.010

Table 6: Invocation costs for the services used in CS2.

3.3 Method of Investigation

In order to answer RQ1, we applied our minimisation technique to both of the case studies. Initially, we tried to randomly generate a test suite that achieves 100% branch coverage for each case study. However, we experienced two issues while using random test generation. It was found to be ineffective in achieving full coverage in CS2 and many of the generated test cases cover exactly the same branches as another test case (equivalent test cases). For CS1 the test suite reached 100% coverage after 1000 test cases and for CS2, test cases covering 3 of the branches (where two ZIP codes of the same city are required) could not be randomly generated within several thousand tries. As a result, we manually generated 4 test cases that cover the uncovered branches in CS2 and applied a simple test case reduction technique to eliminate equivalent test cases.

The resulting test suite sizes were 50 test cases for CS1 and 100 test cases for CS2. The details of both test cases are presented in Table 7. We applied our approach to both of the test suites and measured the minimisation rates achieved by our approach for two different respects: reduction in the number of test cases and reduction in the test cost for both payment models.

Experiment	Test suite size	Coverage	Service invocations	Cost
CS1	50	100%	29760	$1302.90
CS2	100	100%	693	$212.45

Table 7: Details of the test suites generated for CS1 and CS2.

The main reason for seeking the answer to RQ2 is to investigate our assumption that there is a need for reliable test suite. In order to answer RQ2, we measured the false positives occurred during testing of the four weather information services. In this scenario, a false positive occurs when one of the four services returns a negative response to the given ZIP code which is identified as 'invalid' by USPS service.

In order to answer RQ3, we evaluated the generated pareto fronts from two different aspects. First, we ran greedy, NSGA-II and HNSGA-II[2] with all possible combinations of configurations (both payment plans and both number of objectives) in our approach for both case studies and performed a domination analysis to compare the performance of the algorithms. We also measured the distances between discovered pareto

[2]We ran NSGA-II and HNSGA-II for 10 times in order to statistically evaluate their performance.

fronts to give us a better understanding of the differences between the fronts. The distance metric we used (Υ) which is used for measuring how close the discovered set of solutions are to the optimal pareto front. In order to measure this metric, the minimum Euclidean distance between each of the solution from the generated front and the optimal front are calculated and the average of these distances is used as the distance metric. In this part of the experiment, we used the pareto front generated by HNSGA-II as a reference front and measured the distance of the pareto front generated by NSGA-II to it.

4 Results and Analysis

In this section, we present the results from our experiments, provide answers to the research questions and discuss the threats to the validity of the results from experiments we conducted.

4.1 Results

We analysed the results of test suite reduction after the application of our approach to the initial test suites for both case studies in three different aspects. The first aspect is the reduction in number of test cases while retaining the branch coverage of the subject. As presented in Table 8, our approach achieved 84% reduction in the number of test cases for CS1 and 68% reduction for CS2.

Experiment	Number of Test Cases		
	Test Suite	Our approach	Reduction
CS1	50	8	84%
CS2	100	32	68%

Table 8: Reduction in the number of test cases with the application of our approach. The results from our approach are the minimum number of test cases from the initial test suite that provide 100% branch coverage.

During the experiments, we also found that only 12% of the test cases for CS1 and 34% of the test cases for CS2 in the initial test suites found out to be equivalent test cases. This is an important finding that justifies the diversity of the initial test suites, it also provides evidence for the effectiveness of our approach, since the reduction rates are higher than the equivalent test case rates.

Experiment	Cost (contract based)		
	Test Suite	Our approach	Reduction
CS1	29760	415	98.6%
CS2	693	197	72%

Table 9: Reduction in the number of service invocations with the application of our approach. The number for test suite represents the number of service invocations performed by executing the initial test suite. The results from our approach are the minimum number of invocations necessary to achieve 100% branch coverage.

The second aspect is the reduction in the number of service invocations. This aspect relates to the testing cost when using a contract-based payment plan. Our approach in this case targeted for finding the minimum number of service invocations required to achieve 100% branch coverage. As presented in Table 9, our approach found a solution that reduces the number of invocations from 29760 to 415 for CS1 achieving 98.6% reduction while retaining the branch coverage of the test suite. As for CS2, our approach achieved a 72% reduction in the number of service invocations.

The third aspect is the reduction in the monetary cost of testing. This aspect relates to the testing cost when using a per-use payment plan. Our approach in this scenario seeks to find the set of test cases with the least expensive execution cost and achieves 100% branch coverage. As presented in Table 10, our approach

found a solution that reduces the total cost of testing from $1302.90 to $19.86 for CS1 achieving 98.5% reduction while retaining the branch coverage of the test suite. As for CS2, our approach achieved a 69% reduction in the total cost of testing.

Experiment	Cost (per-use based)		
	Test Suite	Our approach	Reduction
CS1	$1302.90	$19.86	98.5%
CS2	$212.45	$65.99	69%

Table 10: Reduction in the number of service invocations with the application of our approach. The number for test suite represents the number of service invocations performed by executing the initial test suite. The results from our approach are the minimum number of invocations necessary to achieve 100% branch coverage.

A general assumption was that one might expect our approach to achieve the same or very similar reduction rates for contract-based plan and the per-use plan. However, the results from Table 9 and Table 10 suggest that this assumption might not hold in all cases. According to the results, in the case of CS1, the difference between the reduction rates is minor. However, for CS2 the difference is significantly higher.

In order to answer **RQ2**, we investigated the false positives generated by the 727 ZIP codes which are identified as invalid by the USPS web service. We tested all the weather services with invalid ZIP codes in order to observe whether they cause a false positive. In this scenario, a false positive occurs when a service does not return a weather information for an invalid ZIP code.

Service	**Type**	**False Positives**	**FP Rate**	**Error Rate**
NOOA.gov	Weather service	232	0.32	0.035
Myweather2.com	Weather service	1	0.0014	0.013
CYDNE.com	Weather service	493	0.68	0.101
Weatherbug.com	Weather service	196	0.27	0.046

Table 11: The list of false positives caused by erroneous test inputs. In this scenario, a false positive occurs when one of the four services return a negative response to the given ZIP code which is identified as 'invalid' by USPS service. Values in 'FP rate' column represent the false positive generation rate for invalid inputs. The values in 'Error rate' column represent the erroneous output generation rates for the test cases in the test suite.

As it is presented in Table 11, there is a big variance in the number of false positives generated by the weather services. We believe this variance might be caused by weather services having ZIP code databases (or using an external ZIP code service) from different sources. For example, 'MyWeather2' service might be using a database which is very similar to the service (Webservicesx) we used in generating the ZIP codes. However, for the other weather services, we observed a much higher false positive rate. For example, for the 'NOAA' service, the tester needs to check the test results for 232 ZIP codes and verify each of these ZIP codes if they used this test suite.

There is also another interesting result observed in Table 11: an invalid ZIP code is more likely to generate a false positive than a valid one to cause an erroneous output. For all services except one, the false positive rate is much higher than error rate.

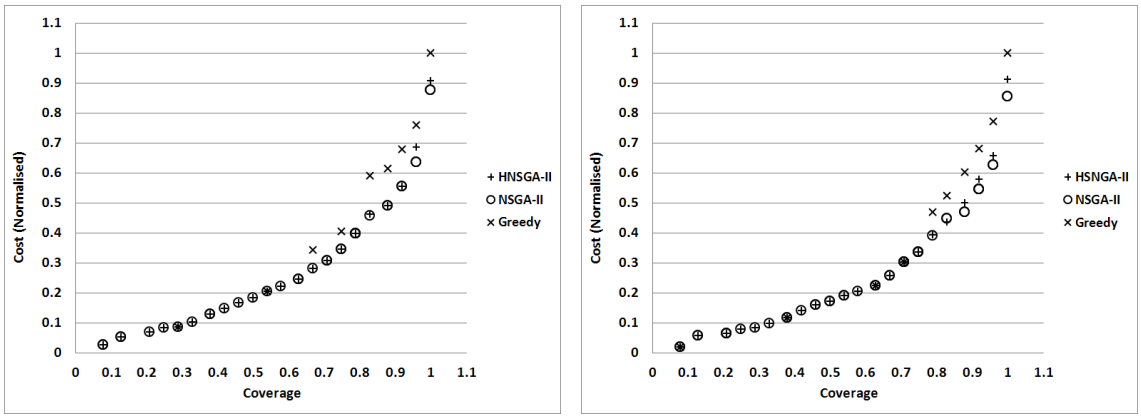
One other important finding we need to discuss in this analysis is that none of the services generated the all 727 false positives we expected to observe. Only 561 of the invalid ZIP codes caused a false positive and for the remainder of the codes the weather services returned a valid response. We believe this is a further indication that supports our claim regarding weather services not using up-to-date ZIP code databases/services.

The results from the domination analysis (presented in Table 12) revealed results that conform with the

Experiment	Contract					Per Use				
	n	HNSGA-II		NSGA-II		n	HNSGA-II		NSGA-II	
		Avg.	σ	Avg.	σ		Avg.	σ	Avg.	σ
CS1 (2 obj.)	23.0	0.5	0.67	1.5	1.43	23.0	0.1	0.3	1.0	1.48
CS2 (2 obj.)	32.7	22.2	1.89	0.3	0.46	32.3	25.1	2.98	0	0
CS1 (3 obj.)	201.8	14.8	3.68	25.1	7.88	204.6	15.8	4.29	23.6	4.03
CS2 (3 obj.)	238.9	212	6.03	1.0	1.18	241	209	10.14	0.7	0.71

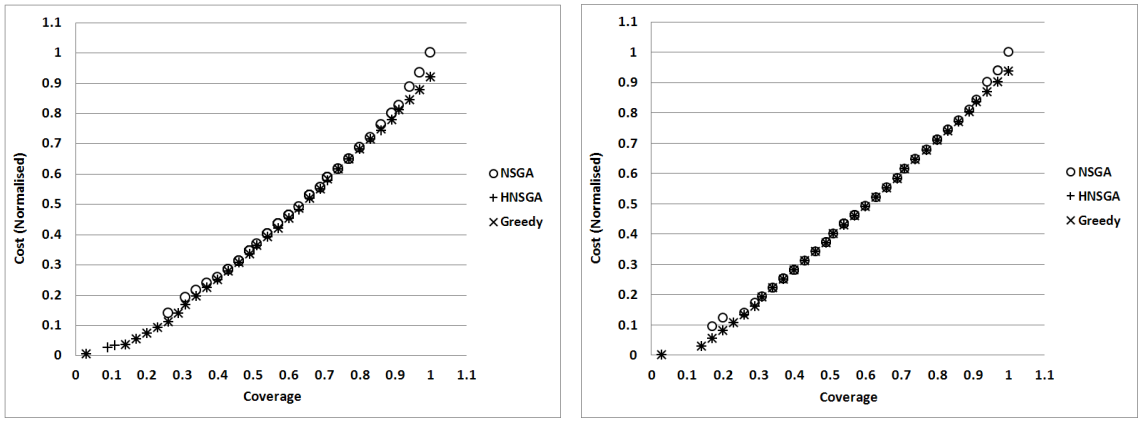
Table 12: Results from our domination analysis. The column 'n' represents the average size of the discovered pareto fronts. 'Avg.' column for each algorithm represents the average number of dominating solutions discovered by the algorithm. The results lead to two important findings that NSGA-II and HNSGA-II's performances are similar for problems in which greedy algorithm does not outperform NSGA-II (such as CS1). However, for problems (such as CS2) where greedy outperforms NSGA-II, HNSGA-II outperforms NSGA-II.

analysis of Yoo and Harman [22]. Our first finding came from the results of CS1 that NSGA-II might outperform HNSGA-II (by a small margin) where additional greedy algorithm cannot outperform NSGA-II (as depicted in Figure 3, 6 and 8). We believe this is due to HSNGA-II starting with a good initial population, leading it to discover solutions around this set whereas NSGA-II explored a larger space using random population. However, as can be observed in Figure 3, 4 5 and 7, the majority of the solutions discovered by both algorithms are the same.



(a) Contract-based payment plan (b) Per-use payment plan

Figure 3: Pareto fronts discovered from the 2 objective optimisation of CS1



(a) Contract-based payment plan (b) Per-use payment plan

Figure 4: Pareto fronts discovered from the 2 objective optimisation of CS2

On the other hand, the results of CS2 provide evidence for the improvements HNSGA-II can provide for problems in which greedy outperforms NSGA-II. The results from both 2- and 3-objective runs indicate that HNSGA-II outperforms NSGA-II by a high margin. For example, on average, 68% of the discovered solutions with 2 objectives for CS2 (contract-based plan) dominate NSGA-II and the domination rate goes up to 89% for the same scenario with 3 objectives. A similar trend was also observed for other configurations of CS2.

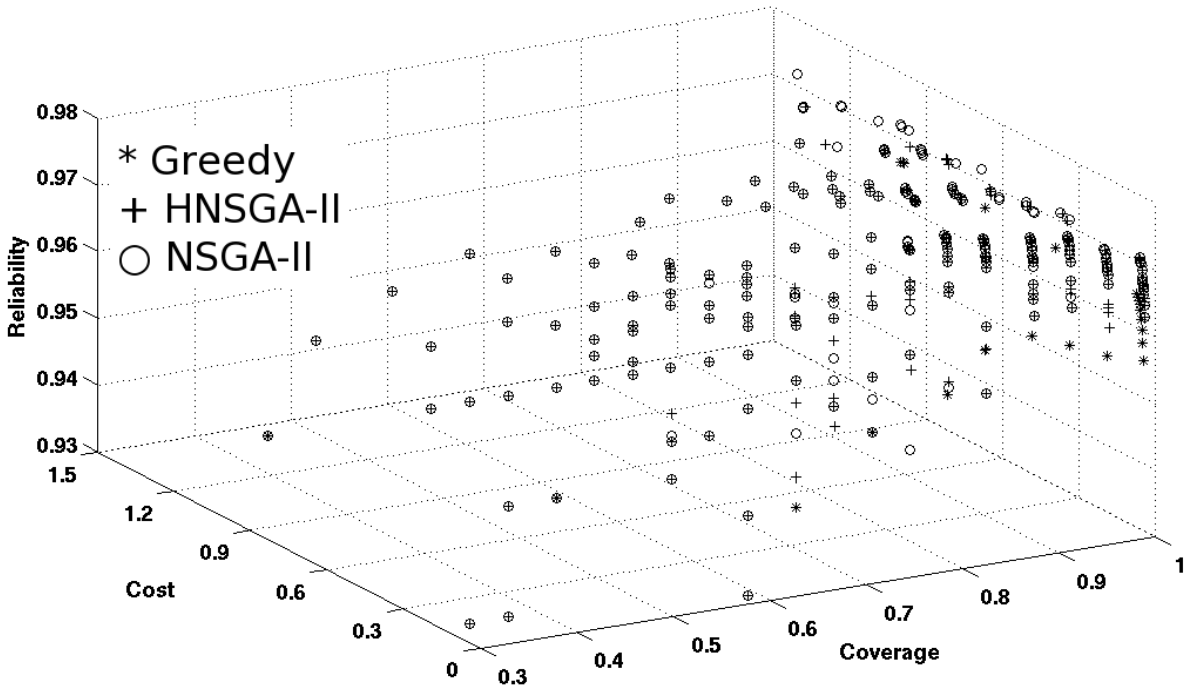


Figure 5: 3 objective optimisation for CS1 with per-use payment plan

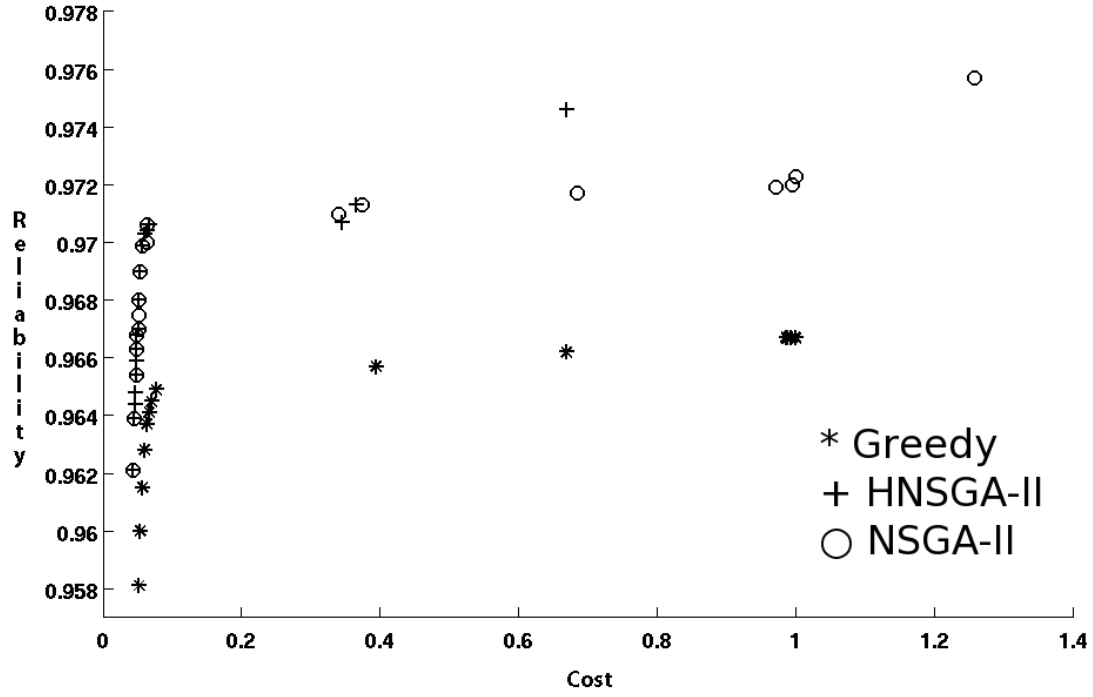


Figure 6: Projected view of Figure 5 focusing on the solutions with 100% coverage score.

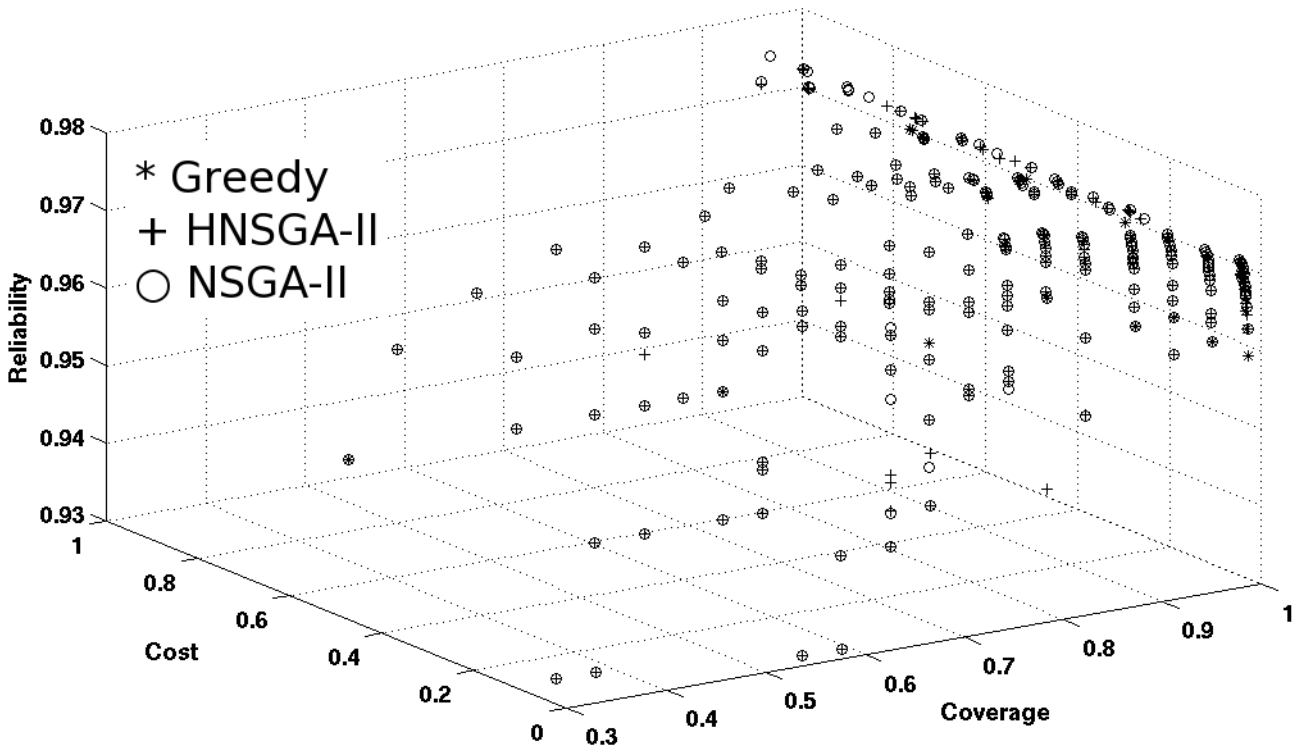


Figure 7: 3 objective optimisation for CS1 with contract-based payment plan

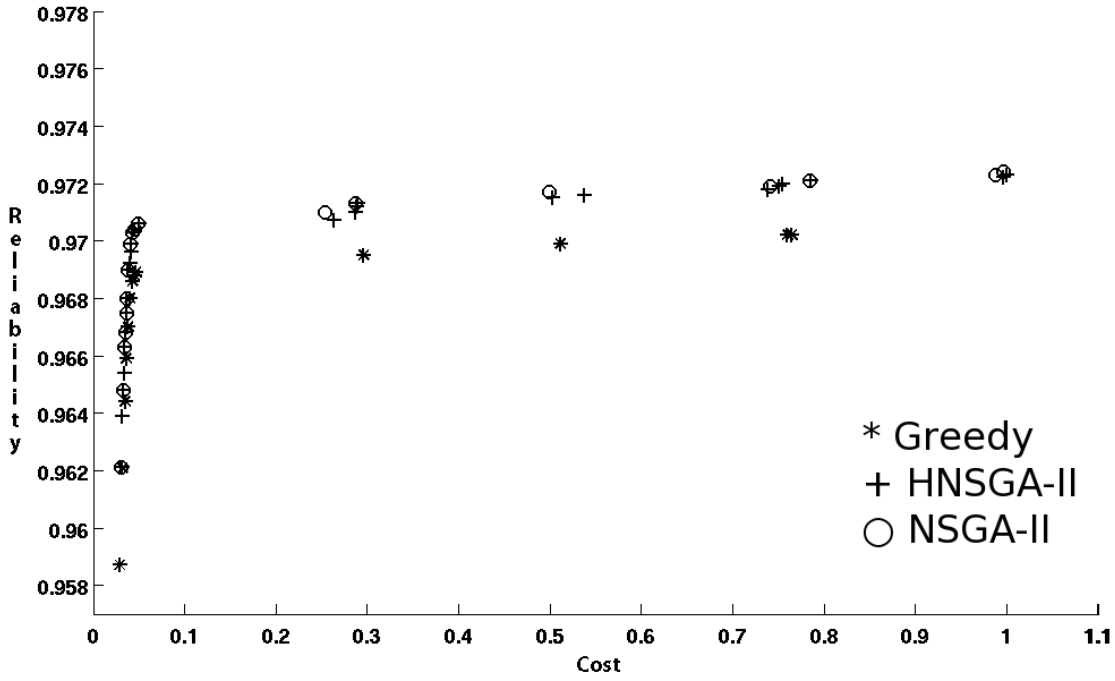


Figure 8: Projected view of Figure 7 focusing on the solutions with 100% coverage score.

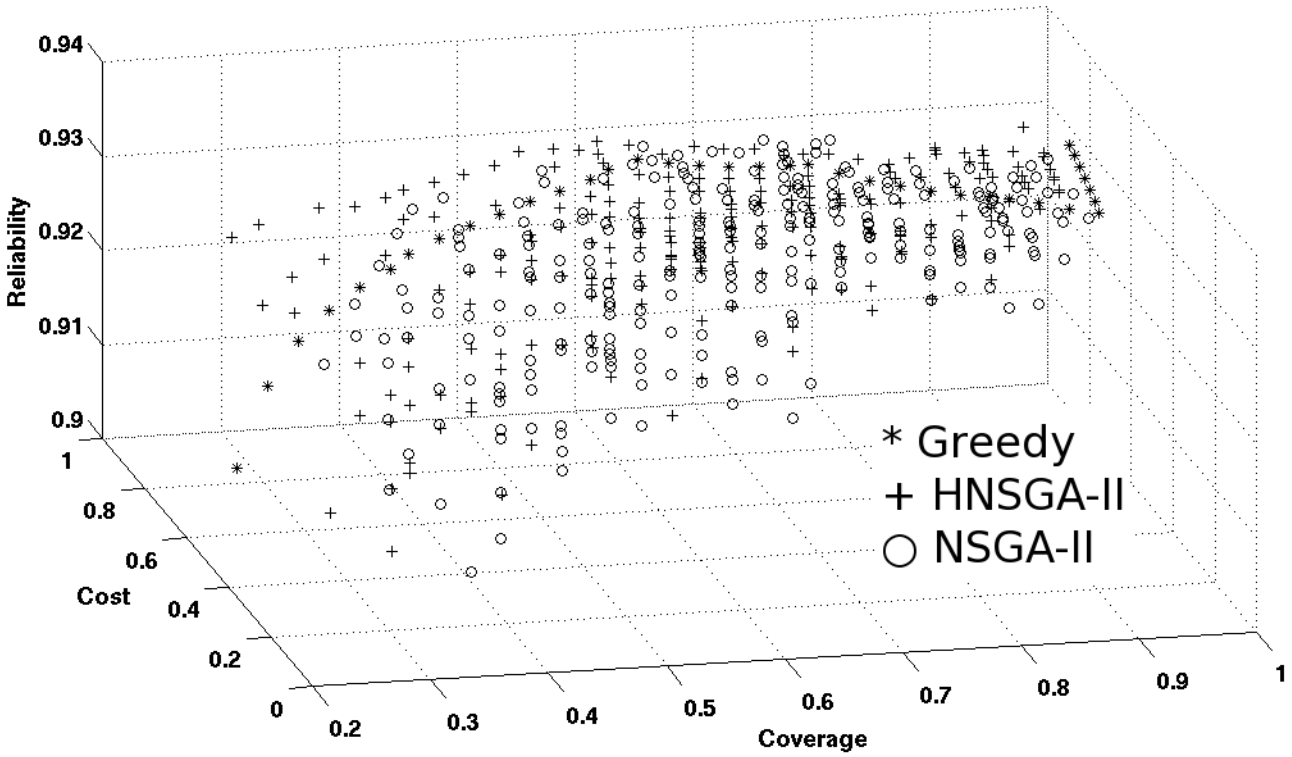


Figure 9: 3 objective optimisation for CS2 with per-use payment plan

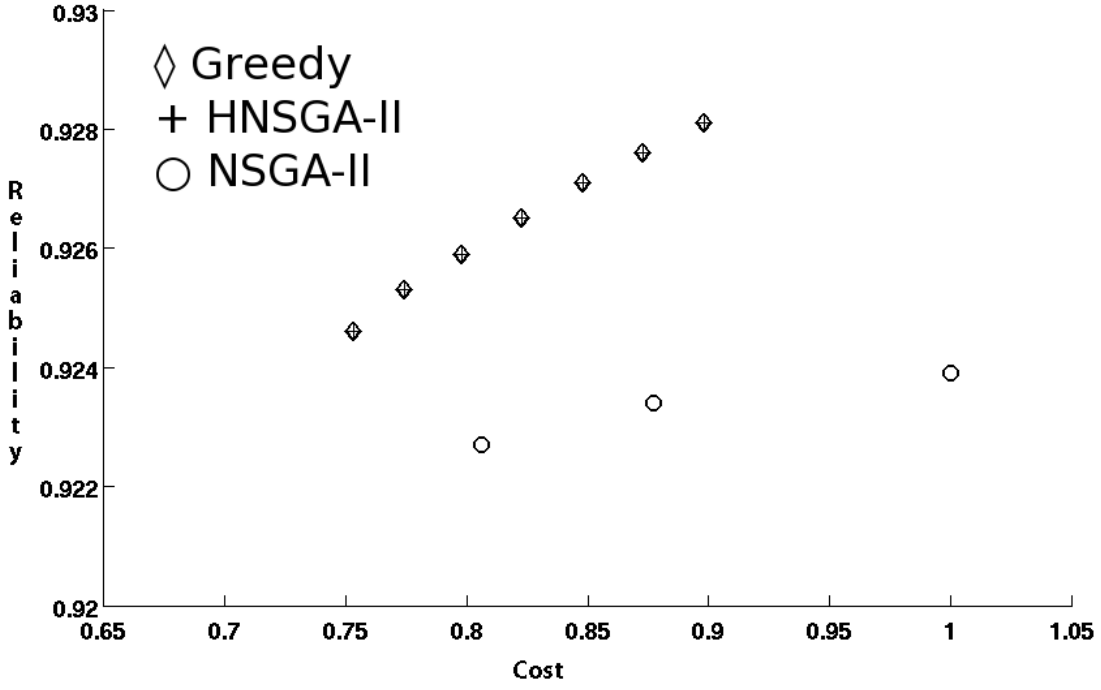


Figure 10: Projected view of Figure 9 focusing on the solutions with 100% coverage score.

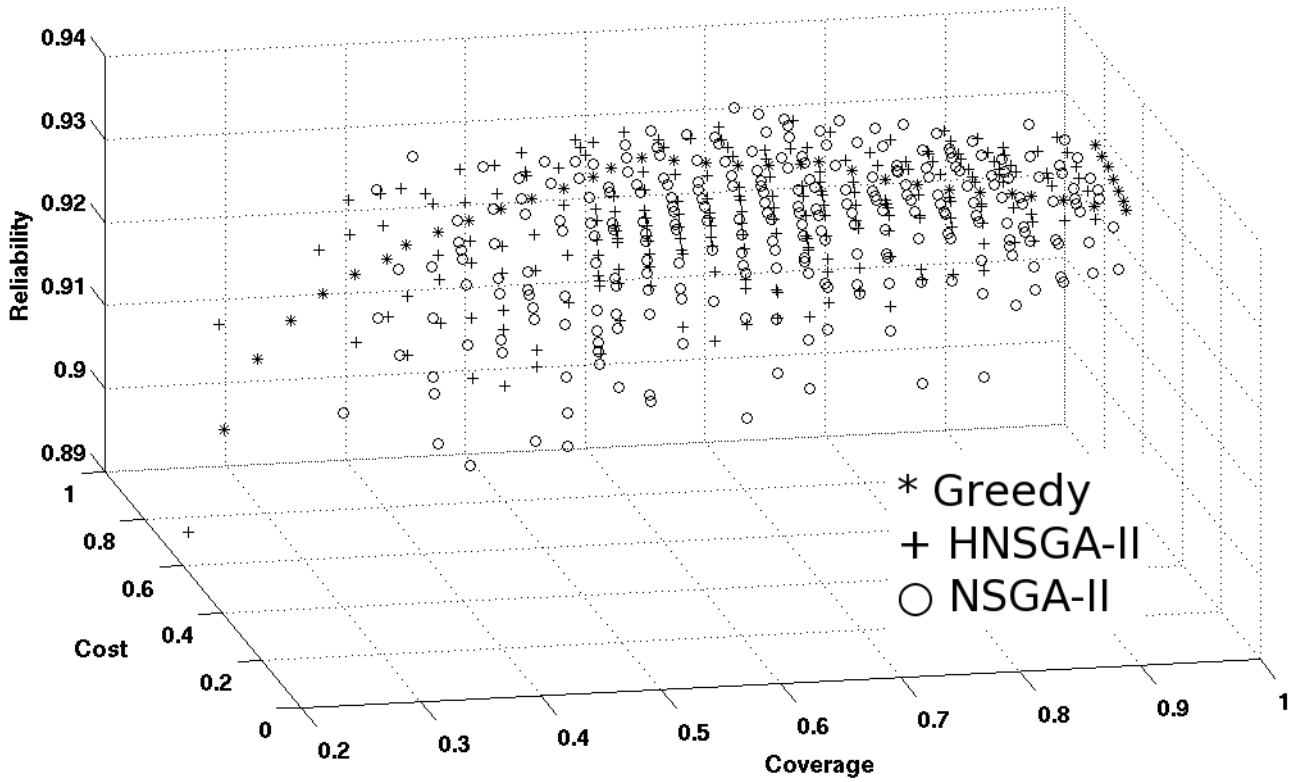


Figure 11: 3 objective optimisation for CS2 with contract-based payment plan

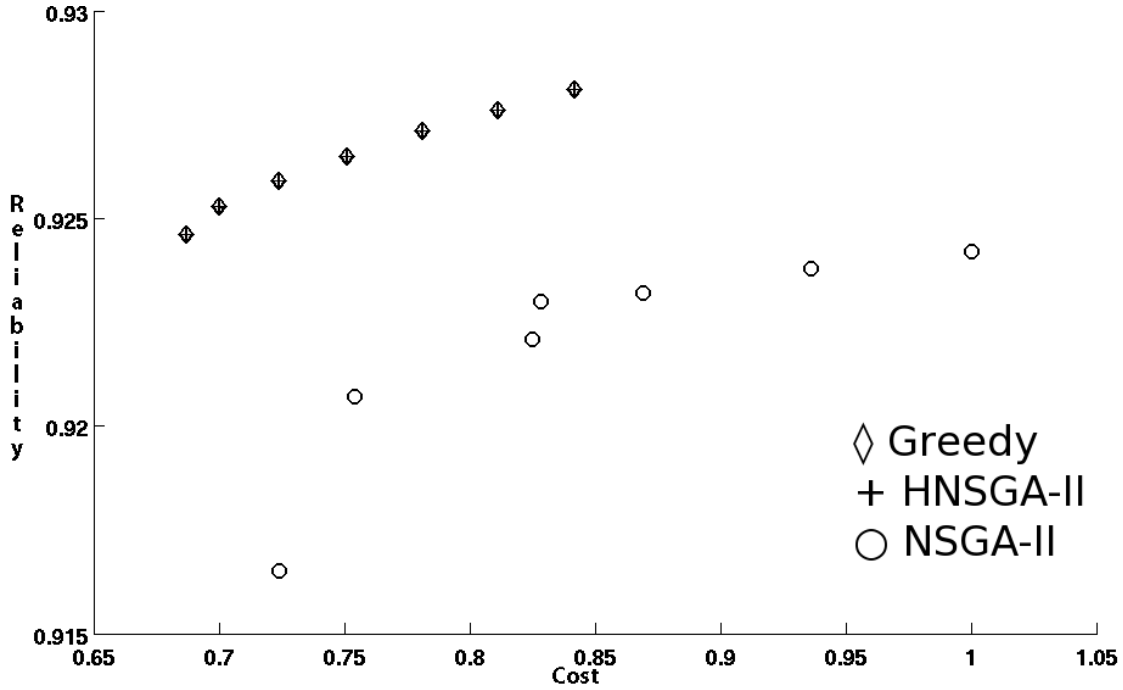


Figure 12: Projected view of Figure 11 focusing around the solutions with 100% coverage score.

The results revealed that the difference between the generated pareto fronts are not as high as presented in Table 13. Before the analysis, we expected the results to have a direct relation with the number of dominant solutions discovered. Our assumption was that the higher the number of dominant solutions, the higher the distance between the pareto fronts must be. The results from the analysis did not validate this assumption. However, the results provide evidence for the fact that for problems in which NSGA-II outperforms greedy,

NSGA-II discovers solutions further from HNSGA-II. This can be observed in Figure 5 and 7 (a closer look of the high coverage areas of these two figures are depicted in Figure 6 and 8), where NSGA-II discovered solutions (especially around 100% coverage) that are far from the solutions discovered by HNSGA-II.

Experiment	Contract		Per Use	
	Average	σ	Average	σ
CS1 (2 objectives)	0.6999	0.71219	0.0353	0.05036
CS2 (2 objectives)	5.4224	1.83578	0.9128	0.24409
CS1 (3 objectives)	4.5524	2.28092	0.07	0.08572
CS2 (3 objectives)	1.521	0.42198	0.4014	0.11026

Table 13: The average distances between the pareto fronts generated by both algorithms. Both algorithms are run 10 times and the average Euclidean distance between pareto fronts are measured using the distance metric Υ.

The distances (between discovered fronts) depicted in figures mentioned here might not accurately represent the actual distance between the solutions due to cost values being normalised. The cost values (for both per-use and contract-based) are normalised in order to fit all discovered fronts into a single graph. However, the distances provided in tables are based on the calculations with the actual cost values and reflect the real distances between fronts.

The results provide evidence for the effectiveness of HSNGA-II over both NSGA-II and greedy algorithm. For example, results from Figure 10 and 12 clearly indicate that HNSGA-II can match greedy's performance for problems in which greedy discovers a pareto front close to optimum front and can outperform NSGA-II. On the other hand, results from Figure 6 and 8 indicate that for problems in which greedy can outperform NSGA-II, HNSGA-II also can outperform greedy by matching NSGA-II's performance.

4.2 Answers to Research Questions

The results from our experiments provide evidence for the effectiveness of our approach in testing cost reduction using multi-objective algorithms and answered **RQ1**. For both case studies, our approach achieved high reduction rates, with up to 84% reduction in the size of test suite and up to 99% reduction in the testing cost (for both payment plans) while retaining the coverage of the initial test suite.

With regards to **RQ2**, the results provide evidence for the fact that invalid inputs have a significant possibility of generating a false positive which might increase the testing cost in ScST. The observed false positive generation rates during the experiments were high (varying between 27% to 68%) for all the services we analysed except for one where the rate was 0.0014. The results also suggest that an invalid input is more likely to cause a false positive than the possibility of a realistic input causing an erroneous output in ScST.

As for **RQ3**, the results provide evidence for the fact that HNSGA-II performance is highly related to the performance of greedy algorithm. Since HNSGA-II executes the same NSGA-II algorithm for problems in which NSGA-II outperforms greedy (such as CS1), it performs similarly to NSGA-II. However, for the problems in which greedy outperforms NSGA-II, we observe the real benefit of HNSGA-II. The evidence for this claim came from the results of CS2, for both payment plans greedy discovered a very good set of solutions and HNSGA-II could not discover better results and resulted with the same pareto front. The advantage of using HNSGA-II is that it can match the performance of the best performing algorithm (out of the two) in any scenario, while outperforming the other one.

5 Conclusion

In this paper, we introduced a solution aimed at reducing the runtime testing cost in ScST by using multi-objective optimisation, and presented an experimental study that investigated the relative effectiveness of proposed approach. We also investigated the HNSGA-II algorithm and compared its effectiveness in ScST

against the other two well known algorithms (NSGA-II and greedy) used in test suite minimisation. In this part of our research agenda, we focused on the cost of runtime testing, branch coverage of the test suite and test suite reliability as our three primary objectives. The results provide evidence for the applicability and the effectiveness of the approach to ScS. Our approach achieved high reduction rates in both case studies with all possible payment plans without reducing the coverage of the test suite. The results also affirm the benefits of using HNSHA-II over NSGA-II especially for certain problems.

5.1 References

References

[1] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," in *SOSE '11: Proceedings of the 2011 IEEE Symposium on Service-Oriented System Engineering*. Irvine, CA, USA: IEEE Computer Society, December 2011, pp. 13–24.

[2] M. Bozkurt and M. Harman, "Optimised realistic test input generation using web services," in *SSBSE 2012: Proceedings of the 4rd International Symposium on Search Based Software Engineering*, G. Fraser and J. Teixeira de Souza, Eds., vol. 7515. Riva Del Garda, Italy: Springer Berlin / Heidelberg, September 2012, pp. 105–120.

[3] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing & verification in service-oriented architecture: A survey," *Software Testing, Verification and Reliability (STVR)*, 2012, To Appear. [Online]. Available: http://dx.doi.org/10.1002/stvr.1470

[4] Calculator web service. Accessed: 27.07.2012. [Online]. Available: http://www.html2xml.nl/Services/Calculator/Version1/Calculator.asmx?WSDL

[5] ECJ 20. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/

[6] FedEx Rate Calculator. Accessed: 27.07.2012. [Online]. Available: https://www.fedex.com/ratefinder/home?cc=US&language=en&locId=express

[7] M. Harman, "Making the case for MORTO: Multi objective regression test optimization," in *Regression 2011*, Berlin, Germany, March 2011.

[8] S. Ilieva, V. Pavlov, and I. Manova, "A composable framework for test automation of service-based applications," in *QUATIC '10: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology*. Oporto, Portugal: IEEE Computer Society, 2010, pp. 286–291.

[9] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang, "BPEL4WS unit testing: framework and implementation," in *ICWS '05: Proceedings of the 2005 IEEE International Conference on Web Services*. Orlando, FL, USA: IEEE Computer Society, July 2005, pp. 103–110 vol.1.

[10] S. Mani, V. S. Sinha, S. Sinha, P. Dhoolia, D. Mukherjee, and S. Chakraborty, "Efficient testing of service-oriented applications using semantic service stubs," in *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*. Los Angeles, CA, USA: IEEE Computer Society, July 2009, pp. 197–204.

[11] P. Mayer and D. Lübke, "Towards a BPEL unit testing framework," in *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*. Portland, Maine, USA: ACM, 2006, pp. 33–42.

[12] M. Palomo-Duarte, A. García-Domínguez, I. Medina-Bulo, A. Álvarez Ayllón, and J. Santacruz, "Takuan: A tool for ws-bpel composition testing using dynamic invariant generation." in *ICWE '10: Proceedings of the 10th International Conference on Web Engineering*, Vienna, Austria, July 2010, pp. 531–534.

[13] Random.org integer generator. Accessed: 27.07.2012. [Online]. Available: http://www.random.org/clients/http/

[14] Remote Methods. Accessed: 27.03.2012. [Online]. Available: http://www.remotemethods.com/

[15] H. Reza and D. Van Gilst, "A framework for testing RESTful web services," in *ITNG '10: Proceedings of the 7th International Conference on Information Technology: New Generations*. Las Vegas, NV, USA: IEEE Computer Society, April 2010, pp. 216–221.

[16] Shipping Calculator. Accessed: 27.07.2012. [Online]. Available: http://www.unitedstateszipcodes.org/shipping-calculator/

[17] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," in *OOPSLA '11: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM, 2011, pp. 189–206.

[18] UPS Calculate Time and Cost. Accessed: 27.07.2012. [Online]. Available: https://wwwapps.ups.com/ctc/request?loc=en_US

[19] USPS Postage Price Calculator. Accessed: 27.07.2012. [Online]. Available: http://postcalc.usps.com/

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

[21] X. Xiao, "Problem identification for structural test generation: First step towards cooperative developer testing," in *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, May 2011.

[22] S. Yoo and M. Harman, "Using hybrid algorithm for pareto effcient multi-objective test suite minimisation," *Journal of Systems Software*, vol. 83, no. 4, pp. 689–701, April.

[23] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2007)*. ACM Press, July 2007, pp. 140–150.

[24] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: A survey," *Software Testing, Verification, and Reliability*, 2010, *To appear*.