Quota-Constrained Test-Case Prioritization for Regression Testing of Service-Centric Systems

Shan-Shan Hou[1,2], Lu Zhang[1,2,*], Tao Xie[3,*], Jia-Su Sun[1,2]
[1]Key laboratory of High Confidence Software Technologies, Ministry of Education
[2]School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
{houss,zhanglu,sjs}@sei.pku.edu.cn
[3]Department of Computer Science, North Carolina State University,Raleigh, NC 27695
xie@csc.ncsu.edu

Abstract

Test-case prioritization is a typical scenario of regression testing, which plays an important role in software maintenance. With the popularity of Web Services, integrating Web Services to build service-centric systems (SCSs) has attracted attention of many researchers and practitioners. During regression testing, as SCSs may use up constituent Web Services' request quotas (e.g., the upper limit of the number of requests that a user can send to a Web Service during a certain time range), the quota constraint may delay fault exposure and the subsequent debugging. In this paper, we investigate quota-constrained test-case prioritization for SCSs, and propose quota-constrained strategies to maximize testing requirement coverage. We divide the testing time into time slots, and iteratively select and prioritize test cases for each time slot using Integer Linear Programming (ILP). We performed an experimental study on our strategies together with three other strategies, and the results show that with the constraint of request quotas, our strategies can schedule test cases for execution in an order with higher effectiveness in exposing faults and achieving total and additional branch coverage.

1 Introduction

Regression testing plays an important role in software maintenance. With the popularity of Web Services, integrating Web Services to build service-centric systems (SCSs) has become an important approach of software development. As a result, testing (including regression testing) of Web Services and SCSs have become a research focus in software testing [2, 3, 10, 12, 16, 18, 19, 22].

Recently, Canfora and Di Penta [4] pointed out that the challenges of testing (including regression testing) of Web Services and SCSs mainly lie in unique features of Web Services, which are "on-line" software artifacts running on the side of their providers. Thus, testers, who cannot own the entities of Web Services, can request Web Services only through the Internet. In testing frameworks of SCSs, Li et al. [10] and Tsai et al. [17] proposed "off-line" testing, which simulates constituent Web Services instead of actually invoking them during testing. With the cost of constructing simulation environments and the risk of losing precision in simulating real environments, "off-line" testing of SCSs can avoid the uncontrollability brought by constituent Web Services. Without inducing extra cost of simulation and losing precision, another feasible way is to actually invoke constituent Web Services during testing SCSs. We call this type of SCS testing as "on-line" testing.

For "on-line" testing, any constraints on the use of Web Services imposed by the service providers may impact the testing process. For instance, superfluous requests to Web Services may bring heavy burden to the network, software, and hardware of service providers, and even disturb service users' normal requests. In an extreme case, if the service provider allows massive vicious requests to a Web Service within a short time, the requests may congest the network or even crash the service's server. To address this issue, providers of Web Services often impose some constraints to avoid responding to superfluous Web Services requests, such as constraints of network flux, storage usage, and request quotas. The request quota is a typical constraint, which defines the upper limit of the number of requests that a user is permitted to send to a Web Service during a certain time range. If a user continues invoking a Web Service with requests after running out of the request quota, the requests will be ignored. Request quotas of some popular Web Services are listed in Table 1.

In this paper, we consider the impact of request quotas on "on-line" regression testing of SCSs. Regression testing is

[*]Corresponding author

Table 1. Request Quotas of Web Services

Web Services	Request Quota
Amazon Historical Pricing Web Service	60000 requests per user per month
eBay Shopping Web Service	5000 requests per IP per day
Yahoo! Web Search Web Services	5000 queries per IP per day per API[1]
Google SOAP Search API	1000 queries per license key per day

often very time-consuming. For instance, the industrial collaborators of Elbaum et al. [7] reported that it costs seven weeks to execute the test suite of one of their products. To accelerate the regression testing process, testers need to execute the test cases as intensively as possible. The intensive execution in regression testing of SCSs may induce superfluous requests to their constituent Web Services. As a result, when testing SCSs, if certain constituent Web Services run out of their request quotas, the execution of the remaining test cases will be postponed. Therefore, fault exposure and the subsequent debugging will also be delayed, and thus the incurred cost will be increased.

To alleviate the impact of request quotas on regression testing of SCSs, we propose to consider request quotas in test-case prioritization. Below we adapt the definition of the traditional test-case prioritization problem [15] to define the problem of quota-constrained test-case prioritization for regression testing of SCSs.

Definition 1: Quota-constrained test-case prioritization problem:
Given:

1. A service-centric system, denoted as S;
2. A test suite for S, denoted as $T = \{tc_i\}$, where $1 \leq i \leq m$;
3. A group of Web Services that are constituents of S, denoted as $WS = \{ws_j\}$, where $1 \leq j \leq n$;
4. The set of requirements for testing, such as code coverage, denoted as $R = \{r_k\}$, where $1 \leq k \leq l$;
5. The request quotas of WS, denoted as $WSQ = \{(q_j, t_j)\}$, where $1 \leq j \leq n$. t_j is a time range and q_j is the maximal number of requests to ws_j that can be processed during each period of t_j;
6. The Web Service invocation matrix, denoted as $TW = (tw_{ij})$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. tw_{ij} is the number of requests sent to ws_j when executing test case tc_i;
7. The testing requirement coverage matrix, denoted as $TR = (tr_{ik})$, where $1 \leq i \leq m$ and $1 \leq k \leq l$. We define tr_{ik} in Formula 1 as below.

$$tr_{ik} = \begin{cases} 1, & \text{if } tc_i \text{ can cover } r_k, \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

8. The set of all the different arrangements of all the test cases in T that can satisfy the quota constraint, denoted as PT;
9. A cost function, denoted as f, that can be applied to any test-case arrangement in PT to yield a cost value for that arrangement;

Problem:
Find T' in PT, such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \leq f(T'')]$.

In this paper, we propose an approach to the problem defined in Definition 1. The basic idea is to extend the traditional total and additional strategies in test-case prioritization [14, 15] to accomodate the constraint imposed by request quotas. In traditional test-case prioritization, test cases are selected one by one to maximize total or additional testing requirement coverage during each step. As request quotas are constraints over a time range, our approach needs to select subsets of test cases that can satisfy the constraint imposed by the request quotas for a period of time. Like traditional test-case prioritization, the aim of our test-case selection is also to maximize total or additional testing requirement coverage. To evaluate our approach, we performed an experimental study by applying our approach for regression testing of a system that integrates Web Services with request quotas. The results demonstrate the effectiveness of our approach.

The rest of this paper is organized as follows. Section 2 shows an example of our approach. Section 3 discusses related research. Section 4 presents the details of our approach. Section 5 describes an experimental study. Section 6 further discusses some issues in our research. Section 7 concludes this paper.

2 Example

In this section, we use a simple stock-trading system (*SSTS*) to illustrate how our approach can help regression testing constrained by request quotas. *SSTS* is constructed through integrating four Web Services, which are provided by stock exchanges and telecommunication companies:
stock_sell: sell stocks at the given price;
stock_buy: buy stocks at the given price;
get_price: get the current price of a stock;
trade_info: send users a message about a successful trade;

Using *SSTS*, a user can customize a list of stocks that he is interested in, point out which stocks he intends to sell or buy, and set his anticipated deal price for each stock in the list. *SSTS* monitors prices of the stocks in the user's list. If the price of a stock that the user intends to sell rises up to

[1]Yahoo! Web Search Web Services include four Web Services, and an API means a Web Service in this context.

```
class Stock {
    public string code;
    public float anticipated_price;
    public Boolean is_sell;
}
void stock_trade (Stock[] s){
    for (int i=0; i < s.length; i++) {
      if ( s[i].is_sell==true &&
          get_price(s[i].code) ≥ s[i].anticipated_price ){
        stock_sell;
        trade_info; }
      if ( s[i].is_sell==false &&
          get_price(s[i].code) ≤ s[i].anticipated_price ){
        stock_buy;
        trade_info; }
    }
}
```

Figure 1. Simple Stock-Trading System

the user's anticipated price, the system will sell the stock for the user and send a message to his mobile phone; if the price of a stock that the user intends to buy falls down to the user's anticipated price, the system will buy the stock for the user and also send a message to his mobile phone. Figure 1 illustrates the main body of the simple stock-trading system.

In our example, we consider the coverage of the four branches, which are induced by the "if" statements, but not the conditions in each branch. Thus, the testing requirements of this example are four branches (denoted as b_1,b_2,b_3, and b_4).

Supposing that we use four test cases (denoted as t_1, t_2, t_3, and t_4) and the request quotas of *stock_sell*, *stock_buy*, *get_price*, and *trade_info* are (100, 1 *time unit*), (150, 1 *time unit*), (600, 5 *time units*), and (400, 5 *time units*), respectively, the Web Service invocation matrix TW and testing requirement coverage matrix TR are as follows.

$$TW=\begin{pmatrix} 50 & 40 & 100 & 90 \\ 20 & 120 & 150 & 140 \\ 90 & 0 & 110 & 90 \\ 30 & 0 & 50 & 30 \end{pmatrix} \quad TR=\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

The subfigures (a) and (b) in Figure 2 show the total number of covered branches (the vertical axis) corresponding to the time units (the horizontal axis) for test execution orders (t_1, t_2, t_3, t_4) and (t_1, t_4, t_3, t_2), respectively. For each execution order, test cases that can execute without overflowing request quotas in each time unit are also demonstrated. For example, in subfigure (a), after t_1 is executed in the first time unit, the available quota of $stock\_buy$ is 110. However, t_2 will request $stock\_buy$ 120 times. Therefore, the execution of t_2 is postponed to the second time unit, in which the quota of $stock\_buy$ is reset to 150. Although (t_1, t_2, t_3, t_4) can achieve optimal total branch coverage without quota constraints, the total branch cov-

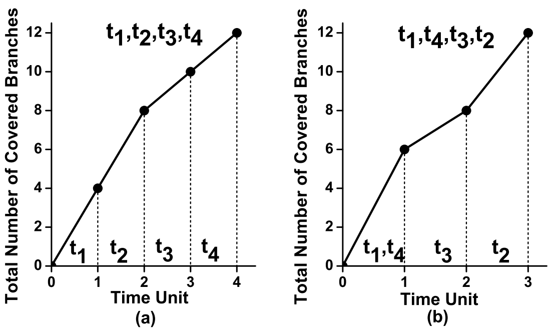

Figure 2. Test Case Execution

erage was delayed when there are quota constraints.

With the preceding request quotas, we can achieve a better execution order (t_1, t_4, t_3, t_2) by selecting as many test cases as possible for each time unit, and prioritize them. In subfigure (b), t_1 and t_4 are selected and prioritized as (t_1, t_4) in the first time unit without overflowing request quotas.

From this example, we can see that a test execution order that can achieve optimal total branch coverage may not achieve its goal when there is a quota constraint. Furthermore, with the constraint of quotas, we can divide the entire regression testing period into a series of time slots and consider test-case selection and prioritization in each time slot. Following this idea, we propose a total strategy for quota-constrained test-case prioritization. Similarly, we propose an additional strategy, which uses a different strategy to select and prioritize test cases in each time slot.

3 Related Work

3.1 Test-Case Prioritization

Test-case prioritization [14] is a typical scenario of regression testing. Rothermel et al. [14, 15] have developed a family of approaches that prioritize test cases to increase their effectiveness of meeting coverage goals at the statement level, such as statement coverage and branch coverage, including total and additional strategies. Elbaum et al. [5] considered coverage goals at the function level, and developed a group of techniques with a coarse granularity, in which they also considered the total and additional strategies.

Recent research mainly focuses on practical issues in test-case prioritization. Considering that the test costs and fault severity may vary in practice, Elbaum et al. [6] and Malishevsky et al. [11] proposed a new metric $APFD_C$, which incorporates different test costs and fault severity, to assess the effectiveness of prioritization. Kim and Porter [9] and Walcott et al. [20] focused on resource-constrained test-case prioritization. They attempted to find out prioritized subsets of the original test suite, which can execute within a given time budget. Kim and Porter [9] proposed an exponential smoothing model, which considers testing history.

Walcott et al. [20] used a genetic algorithm to search for prioritized subsets executed within a time limit.

Some research considers test-case prioritization and test-suite reduction together. Jones and Harrold [8] investigated both test-suite reduction and prioritization based on the modified condition/decision coverage. Both Kim and Porter [9] and Walcott et al. [20] considered reduction in the process of test-case prioritization, as their approaches can acquire a subset from the original test suite and prioritize only the selected test cases.

Different from existing research, in this paper, we consider the impact of request quotas on test-case prioritization in regression testing of SCSs. In fact, the issue considered in this paper comes from the characteristics of the system under test, while issues considered in previous research come from the requirements of the testing process. That is to say, our approach can be complementary to previous approaches. Another main difference is that we rely on Integer Linear Programming (ILP) [21] for test-case prioritization in our approach. Note that, although both our approach and the Bi-criterion approach [1] adopt ILP, there is no straightforward way to extend the Bi-criterion approach to address the issue considered in our approach.

3.2 Regression Testing of Web Services

Some recent research focuses on the regression testing of Web Services. Bruno et al. [3] proposed to use test cases as contracts between service providers and users. Tarhini et al. [16] developed a safe algorithm to select non-redundant test sequences to detect modification-related faults. In this paper, we focus on test-case prioritization in regression testing of SCSs. No previous research on testing Web Services or SCSs has addressed this issue.

4 Quota-Constrained Test-Case Prioritization

In practice, time ranges of request quotas are usually days, weeks, or months, which are much longer than the execution time of one test case. Therefore, we do not consider the execution time in test-case prioritization. In other words, we consider the situation where request quotas are the main constraint in this paper.

As mentioned earlier, the basis of our approach is to extend the traditional total and additional strategies for test-case prioritization. As request quotas for different Web Services are defined on the basis of different time ranges, we need to divide the entire regression testing period into a series of time slots to align these time ranges. As a result, unlike traditional test-case prioritization that selects test cases one by one, our approach selects subsets of test cases for the time slots one by one. To ensure that the selected test cases can satisfy the request quotas for each time slot, we apply the technique of Integer Linear Programming (ILP) [21] to tackle such a test-case selection problem. After selecting a

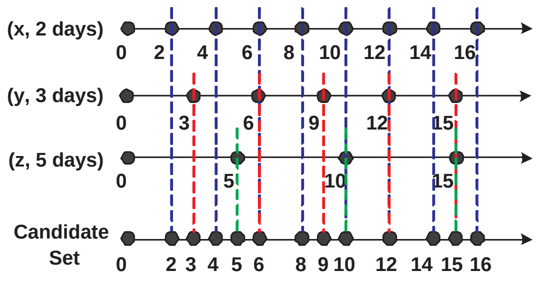

Figure 3. Time Slot Partition

subset of test cases for a given time slot, we can further prioritize the selected test cases using traditional test-case prioritization techniques. After test-case selection and prioritization for one time slot, our approach proceeds to test-case selection and prioritization for the next time slot. Specifically, our approach employs an iterative process including three main steps.

- Step 1: Time-slot partition (Section 4.1). We denote the pth time slot as $ts_p = [bp_p, ep_p]$, which represents the period of time from the *beginning point* bp_p to the *ending point* ep_p. The beginning point of ts_{p+1} is the ending point of ts_p. The aim of time-slot partition is to align different time ranges for different request quotas.
- Step 2: Test-case selection and prioritization for each time slot (Section 4.2). For time slot ts_p, we select test cases that can both satisfy the constraint imposed by request quotas and maximize total or additional testing requirement coverage. After selecting test cases for time slot ts_p, we schedule them using traditional test-case prioritization.
- Step 3: Information refreshing (Section 4.3). As the request quotas, the testing requirement coverage, and the set of remaining selectable test cases for ts_{p+1} depend on which test cases have been selected in ts_p, we need to calculate such information for ts_{p+1} after test-case selection and prioritization for ts_p. We refer to the calculation as information refreshing. After information refreshing, we go back to Step 2 to select and prioritize test cases for ts_{p+1}.

4.1 Time-Slot Partition

The basic rationale of time-slot partition is that each Web Service will not reset its quota for a client within one time-slot range. Therefore, we consider multiples of the range for each Web Service quota. Let us consider the following example. Supposing that the quotas of three Web Services are $\{(x,2 \text{ days}), (y, 3 \text{ days}), (z, 5 \text{ days})\}$, we enumerate the multiples of 2, 3 and 5, and mark them on the candidate set axis in Figure 3. To determine the borders of each time slot, we consider each pair of nesting multiples. Thus, the series of time slots are $[0, 2]$, $[2, 3]$, $[3, 4]$, $[4, 5]$, $[5, 6]$, $[6, 8]$, and so on.

```
Algorithm TS_Partition
Input: $WSQ = \{(q_1, t_1), (q_2, t_2), ..., (q_n, t_n)\}$
        the number of anticipated time slots $s$
Output: $s$ time slots $ts_1$, $ts_2$,..., $ts_s$
Begin
  $ts_1 = [0, ep_1]$, where $ep_1 = min(t_1, t_2, ..., t_n)$
  $p = 1$
  do
    for each $t_j$ in $WSQ$ do
      $cand\_t_j = ((ep_p/t_j) + 1) * t_j$
    end
    $ep_{p+1} = min(cand\_t_1, cand\_t_2, ..., cand\_t_n)$
    $ts_{p+1} = [ep_p, ep_{p+1}]$
    $p++$
  until $p \geq s$
End
```

Figure 4. Algorithm of *TS_Partition*

Formally, we present the algorithm of time-slot partition in Figure 4. The beginning point of the first time slot is 0. The ending point of the first time slot is the smallest value of all the ranges in WSQ (shown in Figure 4). The beginning point of ts_p is the ending point of ts_{p-1}. In order to determine the ending point of the time slot $ts_p = [ep_{p-1}, ep_p]$, we construct a candidate set including time points that are greater than ep_{p-1} and are multiples of time ranges in WSQ. The minimal one in the candidate set is ep_p.

4.2 Test-Case Selection and Prioritization

We iteratively select and prioritize test cases for the time slots. There are two tasks for each time slot: selecting test cases that can maximize the coverage performance goal and execute without overflowing quotas; and prioritizing the selected test cases using traditional test-case prioritization strategies.

Like other test-case prioritization techniques [5, 14], we propose two strategies: the quota-constrained total strategy and the quota-constrained additional strategy. The quota-constrained total strategy selects test cases covering maximal total testing requirements, irrespective of requirement-coverage duplication of different test cases. The quota-constrained additional strategy takes the requirement-coverage duplication into consideration, and selects test cases covering maximal additional not-yet-covered testing requirements. We describe our total and additional strategies in Section 4.2.1 and Section 4.2.2, respectively.

When selecting test cases for a time slot, the basic idea of our strategies is to model our test-case selection as an Integer Linear Programming (ILP) problem [21].

4.2.1 Quota-Constrained Total Strategy

Our quota-constrained total strategy first selects a subset of test cases maximizing testing requirement coverage while satisfying the constraint of request quotas, and then prioritizes selected test cases using the traditional total strategy [14, 15].

As selected test cases can consume request quotas, the request quotas vary in different time slots. Thus, we use $AWSQ_p = \{awsq_{pj}\}$ $(1 \leq j \leq n)$ to denote the available request quota in time slot ts_p. The details of our ILP model for time slot ts_p are described as follows.

(1)The Decision Variables

The decision variables of the ILP model are a $Boolean$ vector, denoted as $X = (x_i)$, where $1 \leq i \leq m$. Variable x_i represents whether test case tc_i is selected in time slot ts_p. Specifically, we define x_i in Formula 2 as below.

$$x_i = \begin{cases} 1, & \text{if } tc_i \text{ is selected to execute in } ts_p, \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

For the example in Section 2, we select $\{t_1, t_4\}$ in time slot $[0, 1]$. Thus, we use $(1, 0, 0, 1)$ to denote this situation.

(2)The Constraint System

The constraint system of this model is a group of inequalities, each of which describes the request quota of a Web Service. The jth inequality indicates that in the time slot, the number of the requests sent to Web Service ws_j when executing the selected test cases should not be over its available request quota. Formally, we describe these inequalities in Formula 3 as below.

$$\sum_{i=1}^{m} tw_{ij} * x_i \leq awsq_{pj} \tag{3}$$

In our example in Section 2, for time slot $[0, 1]$, the constraint system of this selection is in Formula 4 as below.

$$\begin{cases} 50x_1 + 20x_2 + 90x_3 + 30x_4 \leq 100 \\ 40x_1 + 120x_2 \leq 150 \\ 100x_1 + 150x_2 + 110x_3 + 50x_4 \leq 600 \\ 90x_1 + 140x_2 + 90x_3 + 30x_4 \leq 400 \end{cases} \tag{4}$$

(3)The Objective Function

In the time slot, the goal of our quota-constrained total strategy is to maximize testing requirement coverage without considering the duplication of coverage among test cases. Therefore, we define the objective function to yield the maximal testing requirement coverage for the selected test cases. The objective function is in Formula 5 as below.

$$\max \sum_{i=1}^{m} (\sum_{k=1}^{l} tr_{ik}) * x_i \tag{5}$$

In the objective function, the coefficient of variable x_i is $\sum_{k=1}^{l} tr_{ik}$, which represents the sum of the testing requirements covered by test case tc_i. For the example in Section

2, the objective function for time slot $[0, 1]$ is depicted in Formula 6 as below.

$$\max\ (4x_1 + 4x_2 + 2x_3 + 2x_4) \tag{6}$$

By solving the preceding ILP model, we can select a subset of test cases. Then our quota-constrained total strategy prioritizes the selected test cases using the traditional total strategy. For the example in Section 2, testing requirement coverage is actually branch coverage. Therefore, our approach selects $\{t_1, t_4\}$ for the first time slot, and further prioritizes them as the order of running t_4 after t_1.

4.2.2 Quota-Constrained Additional Strategy

Our quota-constrained additional strategy first selects a subset of test cases that can maximize covered testing requirements while satisfying the constraint of request quotas, and then prioritizes the selected test cases using the traditional additional strategy [14, 15]. Specifically, we model test-case selection of the quota-constraint additional strategy in a time slot ts_p as follows.

(1)The Decision Variables

Beside the $Boolean$ vector $X = (x_i)$ introduced in Section 4.2.1, we further introduce another $Boolean$ vector denoted as $Y = (y_k)$, where $1 \leq k \leq l$. Variable y_k ($1 \leq k \leq l$) represents whether test cases selected in the time slot can cover testing requirement r_k. Specifically, we define y_k in Formula 7 as below.

$$y_k = \begin{cases} 1, & \text{if selected test cases in } ts_p \text{ cover } r_k, \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

For the example in Section 2, if t_1 is selected according to the preceding model for time slot $[0, 1]$, the corresponding vectors are $X = (1, 0, 0, 0)$ and $Y = (1, 1, 1, 1)$.

(2)The Constraint System

The constraint system consists of two groups of inequalities. Similar to the quota-constrained total strategy, each inequality in the first group describes the request quota of a Web Service. Thus, we use the inequalities in Formula 3 as the first group.

In the second group, the kth inequality indicates that if the kth testing requirement is satisfied, at least a test case that can cover r_k is selected. Formally, we define the second group of inequalities in Formula 8 as below.

$$\sum_{i=1}^{m} tr_{ik} * x_i \geq y_k \tag{8}$$

For the example in section 2, the second group of inequalities for time slot $[0, 1]$ is in Formula 9 as below.

$$\begin{cases} x_1 + x_2 + x_3 + x_4 \geq y_1 \\ x_1 + x_2 \geq y_2 \\ x_1 + x_2 \geq y_3 \\ x_1 + x_2 + x_3 + x_4 \geq y_4 \end{cases} \tag{9}$$

(3)The Objective Function

In the time slot, the goal of our quota-constrained additional strategy is to maximize the number of covered testing requirements no matter how many times each of them is covered. Therefore, we define the objective function to yield the maximal number of covered testing requirements. Specifically, the objective function is in Formula 10 as below.

$$\max \sum_{k=1}^{l} y_k \tag{10}$$

In the example in Section 2, the objective function for our quota-constrained additional selection in time slot $[0, 1]$ is in Formula 11 as below.

$$\max\ (y_1 + y_2 + y_3 + y_4) \tag{11}$$

By solving the second ILP model, we can select a subset of test cases for time slot ts_p. For the example in Section 2, our approach selects $\{t_1\}$, which can cover all the four branches.

Note that, in time slot ts_p, although the test cases selected according the preceding model can achieve maximal testing requirement coverage, they may not use up the request quotas for time slot ts_p. Under the constraint of the remaining request quotas, we may select more test cases, which cannot increase testing requirement coverage but may be helpful for exposing faults.

To deal with this issue, we continue to select more test cases under the constraint of the remaining request quotas. We refer to this selection as the second-phase selection and the preceding selection as the first-phase selection. As the maximal testing requirement coverage has been achieved, the objective of the second-phase selection should be based on a strategy other than maximizing the number of covered testing requirements. We resort to the total strategy for the second-phase selection, and we model this selection problem as an ILP problem similar to the one described in Section 4.2.1. The main differences are as follows. First, the available Web Service quotas are not those for time slot ts_p, but the remaining request quotas after subtracting the quotas used by test cases selected in the first phase. Second, in the second phase, we no longer consider test cases already selected in the first phase.

For the example in Section 2, $\{t_1\}$ is selected in the first phase for time slot $[0, 1]$ and the remaining quota is (50, 110, 500, 310). In the second-phase selection, $\{t_4\}$ is selected.

After the two phases of selection, we further prioritize all selected test cases using traditional additional strategies [14, 15].

4.3 Information Refreshing

After test-case selection and prioritization in time slot ts_p, our approach considers time slot ts_{p+1}. To acquire the input information for time slot ts_{p+1}, we need to refresh the following information for our ILP models. First, in time slot ts_{p+1}, we no longer consider test cases already selected in time slot ts_p and previous time slots. Second, for the first-phase selection in the quota-constrained additional strategy, we no longer consider testing requirements that test cases selected in time slot ts_p and previous time slots have covered. Third, in time slot ts_{p+1}, we calculate the available quotas as follows. For Web Service ws_j, whose quota is (q_j, t_j), if the starting point of time slot ts_{p+1} is a multiple of its time range t_j, we set the available quota of ws_j for ts_{p+1} as q_j. This situation denotes that we can use the full quota of ws_j again in ts_{p+1}. If the starting point of time slot ts_{p+1} is not a multiple of the time range of ws_j, the available quota of ws_j for ts_{p+1} is the remaining quota by subtracting the quota used by executing test cases selected in ts_p from the available quota of ts_p. This situation denotes that we can use only the remaining quota of ws_j in ts_{p+1}. For the problem in **Definition 1**, the time complexity of each information refreshing process is $O(m * (n + l))$.

5 Experimental Study

We conducted an experimental study to evaluate the effectiveness of our approach. We applied quota-constrained test-case prioritization on an SCS (Section 5.1), and compared the experimental results of our approach with other approaches (Section 5.2). We further discuss threats to validity (Section 5.3).

5.1 Experimental Setup

5.1.1 Subject

In this experimental study, we use a *Travel Agent* system (abbreviated as *TA*), which can have two kinds of users: users who plan independent travel and users who intend to join tour groups. There are 12 Web Services with 17 methods in *TA*. These Web Services can be classified into the following four groups. The first group provides general services, such as queries for the calendar and the weather. The second group is those published by travel agencies that provide group-trip services, such as queries for tour itineraries, queries for prices, and trip arrangement. The third group is those published by the government of tourism cities, which provides services such as queries for local tourist attractions, queries for hotels, and hotel reservations. The last group are Internet banking services providing balance querying, transition, online payment, and so on. All the preceding Web Services are collected from a graduate Web Services course at Peking University, and they are deployed on web server Tomcat 5.0 with SOAP engine Axis 1.4. We construct *TA* by writing a main program in Java, which has 36 branches and requests the preceding Web Services using the dynamic invocation interface.

In order to evaluate the effectiveness of our approach, we seeded 24 faults in the source code of *TA*'s main program, including wrong constants and variables, and faults in arithmetic, logical, and relational operators.

Table 2. Level Values

Level	L1	L2	L3	L4	L5
Value	10000	20000	30000	40000	50000

5.1.2 Test Suites

We used two test suites in our experimental study. The first test suite was generated by a white-box test generation engine called JUnitFactory[2], which is operated by Agitar Software Inc. JUnitFactory examined the source code of *TA* (excluding the Web Services) to generate test cases. We call this test suite as *TS-1* consisting of 489 test cases. The second one is a black-box test suite, which was randomly generated. We call the second test suite as *TS-2* consisting of 1000 test cases.

In this study, we considered test-case prioritization techniques on the basis of branch coverage. The branch coverage ratios of both *TS-1* and *TS-2* are 100%. The fault-exposure ratios of the faults seeded in *TA* of both *TS-1* and *TS-2* are also 100%.

5.1.3 Request Quotas

As the Web Services used in our experimental study are not commercial Web Services, we have to generate request quotas for them. Because providers of Web Services are in charge of the quotas of their Web Services, the quotas should not be specific to a particular SCS. Therefore, we can generate request quotas without considering how our *TA* system uses the Web Services. To make the generated request quotas as close to reality as possible, we considered the following issues in generating quota for each Web Service.

First, the average quota in our experimental study should be about the same as the average quota of commercial Web Services. Specifically, we define a measure named $level$, which denotes the sum of one-day quotas of the 12 Web Services. As different commercial Web Services may have quite different quotas, we considered five $levels$ of one-day overall quotas in our experimental study. Table 2 lists the values for the five $levels$. The average one-day quota for one Web Service in our experimental study ranges from 800 to 4000.

Second, we considered the evenness of the distribution when generating request quotas for each $level$ of overall quotas. Supposing that for a certain $level$ value, the one-day quotas of the 12 Web Services are q_1, q_2, ... q_{12}, a measure $unevenness$ is defined as the ratio between $max(q_1, ..., q_{12})$ to $min(q_1, ..., q_{12})$. In our experimental study, for each $level$, we generated four groups of request quotas with different $unevenness$ values 2, 4, 6, and 8.

[2]`http://www.junitfactory.com/`

Third, as request quotas of Web Services are mainly defined on the basis of days, weeks or months, we randomly chose 1, 7, or 30 as the time range for a Web Service. For a Web Service whose time range is 7 or 30 days, we used 7 or 30 times of the generated one-day quota as its quota for its time range.

5.1.4 Studied Approaches

As our experimental study is based on branch coverage, we considered the following five approaches: quota-constrained total branch (QCTB) strategy, quota-constrained additional branch (QCAB) strategy, traditional total branch (TB) strategy, traditional additional branch (AB) strategy, and random prioritization (abbreviated as Ran). The details of TB and AB strategies are described elsewhere [5, 14] . We employed IBM's SYMPHONY [13] for solving the modeled ILP problems.

5.2 Results and Analysis

As mentioned in Section 5.1.3, for each $level$ value, we generated four groups of request quotas with four different $unevenness$ values. In this study, the experimental result for each $level$ is actually the average result of the four groups of request quotas. Furthermore, in random prioritization, we randomly generated 10 test execution orders for each test suite. For each quota, the result of Ran is the average value of the 10 execution orders.

5.2.1 Effectiveness of Achieving Branch Coverage

In traditional test-case prioritization, the aims of the total and additional branch strategies are to always achieve maximal total and additional branch coverage during the testing. However, when there is a constraint imposed by request quotas, these two strategies may not achieve their goals.

Figure 5 shows the comparison results between the traditional total branch (TB) strategy and our quota-constrained total branch (QCTB) strategy on the two test suites. In Figure 5, the horizontal axis denotes days of regression testing, and the vertical axis denotes the total branch coverage for each day, which is the ratio of the branch coverage of the executed test cases to that of the entire test suite. From the figure, we can see that with the same level (denoted as the same symbol on lines), the total branch coverage of our QCTB (denoted as a solid line) is always higher than that of TB (denoted as a dash line) for each test suite. The result indicates that our QCTB strategy is a better strategy for achieving total branch coverage under quota constraints than the traditional TB strategy.

In Figure 5, the total branch coverage of most strategies increases very slowly on the 7th and 30th days. Some portions of the coverage lines are almost plateaus, such as QCTB-L5 and TB-L3 for *TS-2* on the 7th day. We suspect that it is because some request quotas' time ranges are defined as 7 and 30 days in Section 5.1.3. Such request quotas probably have been used up by test cases selected during the

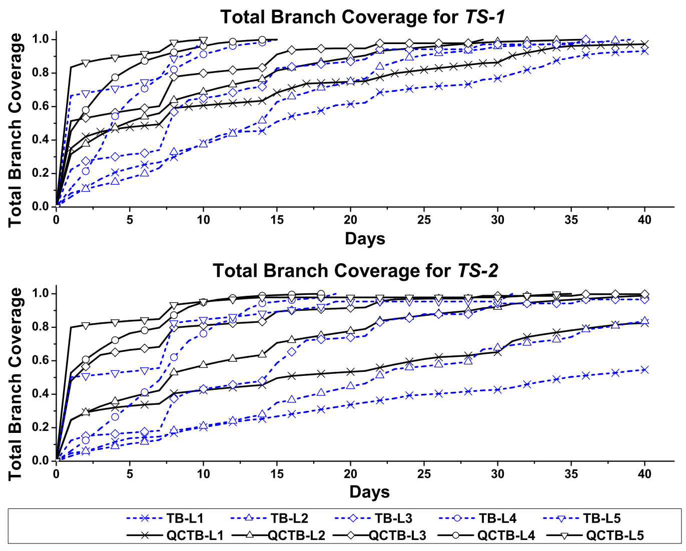

Figure 5. Total Branch Coverage

first several days of the time ranges. Therefore, on the 7th and 30th days, which are the last days of the time ranges, few test cases are selected due to the lack of request quotas.

For the comparison between the traditional additional branch (AB) strategy and our quota-constrained additional branch (QCAB) strategy, the result is similar to the preceding comparison between TB and QCTB. Our QCAB strategy always outperforms the traditional AB strategy for each test suite and each quota level. Due to space limit, we omit the figure that shows the comparison results between AB and QCAB.

5.2.2 APFD

The average percentage of faults detected (APFD) metric [15] is a widely adopted metric in evaluating test-case prioritization techniques. This metric is based on the increase of the number of executed test cases. When the time required for executing a test suite is long, Elbaum et al. [5,6] propose to calculate the fault exposure ratio with the increase of time, such as days or hours. In this paper, we refer to this APFD metric as the time-based APFD. In our experimental study, the time for executing a test suite can be very long due to the quota constraints. Therefore, we adopt the time-based APFD in our experimental study.

We depict a line to show the fault exposure ratio (vertical axis) versus the fraction of the time consumed (horizonal axis), and define the APFD value as the area enclosed by the line and the horizonal axis. Supposing that the test suite execution requires m days and the fault exposure ratio on ith day is f_i, we formally present the time-based APFD in Formula 12.

$$\frac{\sum_{i=1}^{m}(f_{i-1}+f_i)}{2m} \tag{12}$$

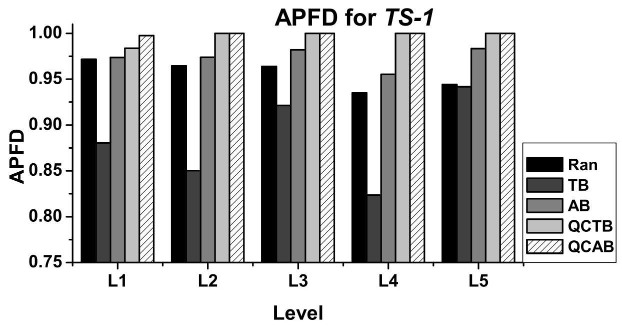

Figure 6. APFD for *TS-1*

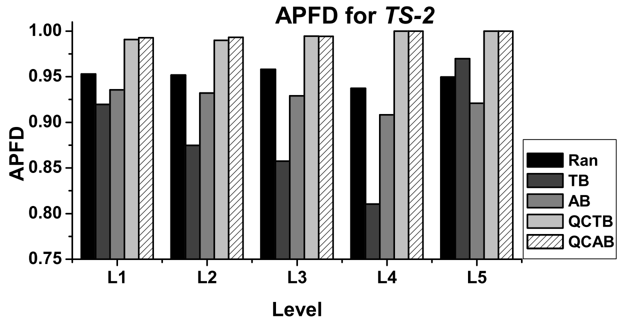

Figure 7. APFD for *TS-2*

Using the preceding APFD metric, we show the results of Ran, TB, AB, QCTB, and QCAB for test suite *TS-1* and *TS-2* in Figures 6 and 7, respectively. The time-based APFD values (vertical axis) for different quota levels (horizonal axis) are illustrated. These figures demonstrate the following trends.

First, both our QCTB and QCAB strategies outperform the other three strategies for both test suites at all the five levels. The APFD values of our QCTB and QCAB are comparable. Compared with Ran, TB, and AB strategies, our QCAB improves APFD values with 4.49%, 11.28%, and 4.83% on average, respectively.

Second, neither TB nor AB can always outperform the Ran strategy for both test suites at all the five levels. We suspect that the reason might be that the quota constraints can affect TB and AB more than the random strategy. The TB strategy always tries to execute test cases covering more branches earlier, and it is very likely to result in executing test cases consuming more request quotas earlier. Therefore, there may be significant delays during the early stages of regression testing.

5.2.3 Time Cost of Test-Case Prioritization

We performed our study on a 2.26G Hz Pentium 4 1GB RAM system running Windows XP Professional. For our QCTB and QCAB, each column of Table 3 lists the average time cost of test-case prioritization in minutes for each level. The result demonstrates the following trends. First, with the relaxing of the quota constraints, the time cost of both our QCTB and QCAB are decreasing. Second, in each test suite, compared with QCTB, the time cost of QCAB is consistently higher. We suspect that it is because QCAB imports more variables and constraints to the ILP model, and has the second-phase selection. Third, compared with *TS-1*, the time cost of *TS-2* is much higher. In fact, the size of *TS-2* is twice the size of *TS-1*, which induces more complex ILP computation with more decision variables and constraints.

Table 3. Average Time Cost of Prioritization

Time (mins)		L1	L2	L3	L4	L5
TS-1	QCTB	7.75	6.67	5.92	2.67	2.01
	QCAB	9.13	8.05	6.50	3.67	3.12
TS-2	QCTB	29.51	16.12	9.13	6.75	5.25
	QCAB	40.22	28.00	9.58	7.25	5.74

5.3 Threats to Validity

In our experimental study, there are three main threats to external validity. First, we use only one subject system. The particular characteristics of the subject system may have effect on our experimental results, which may not be generalized to other SCSs. To reduce this threat, we plan to evaluate our approach using several other SCSs in future work. Second, different quota values may induce various experimental results. To reduce this threat, we used different groups of request quotas with different $level$ and $unevenness$ values, and all these quotas are similar to those of existing commercial Web Services. Third, the subject system and the used Web Services are implemented in Java. Our experimental results may not be generalized to other situations. To reduce this threat, we plan to do more experiments using large-sized SCSs with Web Services running on different platforms.

6 Discussion

6.1 Execution Time of Test Cases

We do not consider the execution time of test cases in our approach. In fact, in situations where the execution time of test cases needs to be considered, we can adjust our models by incorporating into the constraint system another inequality, which denotes that the sum of execution time of the selected test cases should not be longer than the length of the corresponding time slot. In an extreme case, if the request quota of a Web Service is not enough for executing a test case, it is necessary to refactor the test case into several smaller test cases that can satisfy the request quota.

6.2 Limitation

Our approach aims to facilitate regression testing of SCSs that are composed of Web Services with only quota constraints. Besides request quotas, service providers may impose other constraints on the use of Web Services. Therefore, our approach may not be applicable or effective when constraints other than request quotas exist. However, except those Web Services without constraints, request quotas are the most popular constraint for existing Web Services. As a result, our approach should be applicable and effective for many existing SCSs. In future work, we plan to further investigate test-case-prioritization techniques that can deal with regression testing of SCSs integrating Web Services with various kinds of constraints mentioned in Section 1.

7 Conclusion

Testing SCSs may induce plenty and intensive invocations of Web Services. Therefore, request quotas of Web Services can be a big constraint for testing SCSs. In this paper, we proposed quota-constrained test-case prioritization for regression testing of SCSs. In particular, we proposed two strategies that aim at maximizing the total and the additional testing requirement coverage with the constraint of request quotas. In our approach, we divide the testing time into time slots, and iteratively select and prioritize test cases for each time slot using ILP.

We performed an experimental study on our strategies together with the random strategy, the traditional total strategy, and the traditional additional strategy. The results show that compared with other strategies, our new strategies can schedule test cases for execution with higher effectiveness of achieving total or additional branch coverage and exposing faults with the constraint of request quotas.

Acknowledgments. This research is sponsored by the National Basic Research Program of China under Grant No. 2002CB312000, and the High-Tech Research and Development Program of China under Grant No.2006AA01Z189, 2007AA010301, 2006AA01Z156, and 2006AA01Z175, and the National Science Foundation of China No. 90718016. Tao Xie's work is supported in part by NSF grant CCF-0725190.

References

[1] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proc. International Conference on Software Engineering*, pages 106–115, 2004.

[2] J. Bloomberg. Testing web services today and tomorrow. *The Rational Edge E-zine for the Rational Community*, 2002.

[3] M. Bruno, G. Canfora, M. Di Penta, G. Esposito, and V. Mazza. Using test cases as contract to ensure service compliance across releases. In *Proc. International Conference on Service Oriented Computing*, pages 87–100, 2005.

[4] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proc. International Symposium on Software Testing and Analysis*, pages 102–112, 2000.

[6] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. International Conference on Software Engineering*, pages 329–338, 2001.

[7] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[8] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proc. International Conference on Software Maintenance*, pages 92–101, 2001.

[9] J. M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proc. International Conference on Software Engineering*, pages 119–129, 2002.

[10] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. BPEL4WS unit testing: Framework and implementation. In *Proc. International Conference on Web Services*, pages 103–110, 2005.

[11] A. Malishevsky, J. R. Ruthru, G. Rothermel, and S. Elbaum. Cost-cognizant test case prioritization. Technical report, Department Computer Science and Engineering of University of Nebraska, 2006.

[12] A. J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[13] T. Ralphs and M. Guzelsoy. The SYMPHONY callable library for mixed integer programming. In *Proc. INFORMS Computing Society Conference*, pages 61–73, 2005.

[14] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proc. International Conference on Software Maintenance*, pages 179–188, 1999.

[15] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[16] A. Tarhini, H. Fouchal, and N. Mansour. Regression testing web services-based applications. In *Proc. International Conference on Computer Systems and Applications*, pages 163–170, 2006.

[17] W. T. Tsai, C. Fan, and Y. Chen. DDSOS: A dynamic distributed service-oriented simulation framework. In *Proc. Annual Simulation Symposium*, pages 160–167, 2006.

[18] W. T. Tsai, R. Paul, W. Song, and Z. Cao. Coyote: An XML-based framework for web services testing. In *Proc. International Symposium on High Assurance Systems Engineering*, pages 173–174, 2002.

[19] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang. Extending WSDL to facilitate web services testing. In *Proc. International Symposium on High Assurance Systems Engineering*, pages 171–172, 2002.

[20] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time aware test suite prioritization. In *Proc. International Symposium on Software Testing and Analysis*, pages 1–11, 2006.

[21] H. Williams. *Model Building in Mathematical Programming*. John Wiley, New York, 1993.

[22] W. Xu, A. J. Offut, and J. Luo. Testing web services by XML perturbation. In *Proc. International Symposium on Software Reliability Engineering*, pages 257–266, 2005.