Abstract

Service-oriented applications can be expensive to test because services are hosted remotely, are potentially shared among many users, and may have costs associated with their invocation. In this paper, we present an approach for reducing the costs of testing such applications. The key observation underlying our approach is that certain aspects of an application can be tested using locally deployed semantic service stubs, instead of actual remote services. A semantic service stub incorporates some of the service functionality, such as verifying preconditions and generating output messages based on postconditions. We illustrate how semantic stubs can enable the client test suite to be partitioned into subsets, some of which need not be executed using remote services. We also present a case study that demonstrates the feasibility of the approach, and potential cost savings for testing. The main benefits of our approach are that it can (1) reduce the number of test cases that need to be run to invoke remote services, (2) ensure that certain aspects of application functionality are well-tested before service integration occurs.

1 Introduction

Service-oriented computing promises greater flexibility and efficiency in application development by enabling applications to be composed using third-party services; however, such applications can be difficult to test. Several factors, such as multiple runtime configurations, remote hosting of services, lack of access to service source code, and unanticipated changes in service semantics, present challenges in testing service-oriented applications [6]. For example, lack of access to source code prevents the use of code-based testing techniques; lack of control over service evolution can complicate regression testing.

Another significant factor, which we address in this paper, is the cost of testing service-oriented applications. Testing a service-oriented application involves invoking services, which is inherently expensive because services are remotely hosted (therefore, susceptible to network traffic and downtime) and shared by many users (therefore, dependent on server load). Moreover, many services have restrictions on, or costs associated with, their invocation (e.g., a pay-per-use model of service usage). Therefore, running a large test suite incurs costs—both time and monetary—that could be lowered through a better selection of test cases that invoke remote services. Service providers may have an interest in limiting testing activities too. Execution of large test suites, containing many unnecessary and invalid test cases, by a client can lead to wasteful resource consumption and degraded performance, which affects all clients.

Much of the existing research on services testing has focused on developing automation frameworks and techniques for test-data generation (e.g., [1, 13, 18, 22, 26]). Other research has addressed problems such as regression testing (e.g., [3]), contract-based testing (e.g., [9, 23, 25]), and conformance and interoperability (e.g., [2, 11]). However, there has been a limited investigation of how costs of testing service-oriented applications can be reduced.

In this paper, we present a novel approach for lowering the costs of testing service-oriented applications. We address the problem from the service-consumer perspective. The main observation underlying our approach is that certain aspects of testing a service application can be performed using locally deployed, semantic service stubs. A semantic service stub incorporates some of the functionality of the service that it represents. Examples of such functionality include verifying preconditions, and returning appropriate response and exception messages.

Existing research has proposed different types of behavioral contracts, such as preconditions and postconditions, that can be associated with service interfaces [9, 10, 22]. Such contracts are typically used for generating test cases.

\[1\] Execution of unreasonably large test suites may in fact be seen as a denial-of-service attack [5].

\[2\] Reference [6] provides a good overview of the issues in testing of service-centric systems from the perspectives of different stakeholders.
and test oracles. In our approach, we leverage behavioral contracts to generate semantic service stubs that contain code to simulate the behavior specified in the contracts.

To generate semantic stubs, service providers create semantic annotations for the exposed services, which are added to the service source code or XML descriptions. Next, automated translators are used to analyze the annotations and generate the stub source code. In this paper, we illustrate these steps using the Java Modeling Language (JML) [16] to specify service semantics at the source code level, and the JML compiler (JMLC) to generate stub code automatically from the JML specifications.

Service consumers can use such stubs to test their applications before integrating them with the actual remote services. This approach has several benefits. First, test cases that are invalid because they violate service contracts can be detected, and fixed or filtered, before the invocation of remote services. Second, local testing using stubs can reveal faults in the application, which can be fixed to make the application more robust before it is integrated with remote services. Third, because stubs are deployed locally, they can be used early and repeatedly in the application development cycle, without concerns regarding costs of remote service invocations.

We present a classification of test cases for the client application. Using the classification, a client test suite can be partitioned into different categories, such that test cases from some of the categories can be run using local semantic stubs only, and need not be rerun after the application has been integrated with remote services.

To demonstrate the feasibility of our approach and evaluate its benefits, we implemented a prototype and performed two case studies. The results of the studies indicate that our approach can (1) reduce the size of the test suite that needs to be run on remote services, (2) lead to early detection of faults, and (3) result in faster execution of test cases.

The main contributions of the paper are
- A novel approach that uses semantic service stubs to improve the efficiency and effectiveness of testing service-oriented applications
- The results of a case study that demonstrate the feasibility and benefits of the technique

In the next section, we provide an overview of the approach, and illustrate semantic stub generation. Section 3 discusses how semantic stubs can improve the testing of service-oriented applications. Section 4 presents the empirical results. Section 5 discusses related research; finally, Section 6 summarizes our work and lists directions for future research.

2 Semantic Service Stubs

The central feature of our approach is the use of semantic service stubs that incorporate some of the functionality of the actual services, and that can be deployed locally by service consumers for testing their applications. Typical stubs incorporate little or no semantics of the software components that they represent, and therefore, provide no benefit of enabling meaningful testing of the applications from which they are invoked.

A semantic service stub extends the notion of behavioral mock objects [24] to the domain of web services; it is based on design-by-contract principles, in which a formal contract is associated with each software component [19]. The elements in a contract include preconditions that a client of the component must satisfy to obtain the services of the component, and postconditions that the component guarantees to the client. A semantic service stub contains code that verifies whether the inputs to a service operation are valid, and generates appropriate output and exception messages based on postconditions of the operation. Additionally, it can contain code to enforce invocation sequences and state depen-
2.1 Overview of our approach

Figure 1 presents an overview of the approach; it illustrates two scenarios. In the first scenario, service providers generate semantic stubs and sample test data, and make them available for download. To generate semantic stubs, the provider infers semantic annotations for the service. This can be done in a semi-automated manner using program-analysis techniques. A code analyzer can infer potential pre and postconditions from the service source code, which can be manually refined. Next, the provider adds the inferred annotations to the service source code or, alternatively, to the WSDL specifications.

The stub generator consists of a code generator and a test-data generator. There are two alternative sets of inputs to the stub generator. In the first alternative, the stub generator takes as inputs the annotated service source code and the standard WSDL specifications. The code generator automatically generates the stub code from the source-code annotations. The test-data generator parses the WSDL and the source-code annotations to generate test data that are returned if the operations of the service are invoked correctly. In the second alternative, the stub generator takes as input the WSDL specifications with embedded semantic annotations. In this case, the code generator uses the WSDL annotations to generate the stub code.

Service consumers can download the service stubs, along with the test data, and deploy them locally, so that the client applications invoke the service stubs instead of the remote services. Thus, the test cases for a client application can be executed in this environment, without incurring the costs of invoking remote services during the early phases of application development and testing.

In the second scenario, shown at the bottom in Figure 1, the service provider provides only the WSDL with embedded semantic annotations. Service consumers download the WSDL, and use their own custom, or another third-party, stub generator to process the annotated WSDL. Thus, in this scenario, the providers do not host the stubs for download (unlike in the first scenario). This scenario offers flexibility to the consumers in letting them select the stub generator, instead of being restricted to using pre-generated stubs.

As the service code evolves, the stubs need to be updated so that they remain consistent with the service code. Therefore, when a new service version is made available, providers have to ensure that the stubs are regenerated from the new code; moreover, clients that migrate to the new version have to update the stubs as well.

2.2 Semantic annotations

We illustrate semantic annotations using the Java Modeling Language (JML) [16], a behavioral interface specification language for Java. (Other approaches for behavioral specification, such as iContract [15], could be used too.) JML includes an extensive set of constructs for specifying various aspects of a component’s behavior, such as method preconditions, postconditions, and exception conditions, class invariants, loop invariants, etc.\(^3\)

Source code annotations

In JML, preconditions, postconditions, and exception conditions are specified using the requires clause, the ensures clause, and the signals clause, respectively. Figure 2 presents the interface specifications for a sample service, cartAdd, from the Amazon Associate Web Services.\(^4\) Service cartAdd takes as inputs (among others) a cart id and an item id, and adds the item to the cart. The figure lists the preconditions for the input parameters. For example, the preconditions for the first seven parameters state that these parameters can be neither null nor contain

```
public interface AmazonAPISpecification {
    /* @ requires service != null && service != "";
    @ requires AWSAccessKeyId != null &&
    AWSAccessKeyId != "";
    @ requires subscriptionId != null &&
    subscriptionId != "";
    @ requires operation != null && operation != "";
    @ requires ASIN != null && ASIN != "";
    @ requires cartId != null && cartId != "";
    @ requires HMAC != null && HMAC != "";
    @ requires item.qty > 0;
    @ ensures isValid(cartId) =>
    result.indexOf("PurchaseURL") => 1;
    @ ensures isValid(cartId) =>
    result.indexOf("Item") => 1;
    @ ensures isValid(cartId) =>
    result == null;
    */
    public boolean isValid(String id);
}
```

Figure 2. Sample Amazon Associate service annotated with pre, post, and exception conditions using JML.
an empty string. Similarly, the last precondition states that the quantity of the item to be added must be greater than zero. The @ensures clauses for cartAdd() establish constraints on the return value. The clauses state that if the cart id is valid, the returned string definitely contains a substring “PurchaseURL” and a substring “Item”; however, if the cart id is invalid, an empty string is returned.

Discussion
Adding service annotations to either the code or the WSDL requires the service provider to provide this information. It may appear that this places an unreasonable burden on service providers. However, many of the widely used services, such as Google and Amazon services, have extensive text-based documentation that provide service details with respect to required parameters, valid requests, sample requests, exception handling, etc. Intra-enterprise services are typically developed by following the conventional software-development life cycle, in which the design phase formally captures service specifications in models, such as UML. Therefore, the available service documentation and models can be leveraged for creating semantic annotations.

Moreover, automated program-analysis techniques, such as symbolic execution [14], can be used to analyze the service source code to recover potential contracts, such as exception conditions (e.g., [4]). Automatically inferred contracts can assist service providers in creating semantic annotations and ensuring that the annotations are consistent (i.e., do not contradict the contracts present in code) and complete (i.e., do not miss contracts implicit in code).

### 2.3 Stub and test-data generation

After JML annotations have been added to the source code, stubs can be generated automatically using the standard JML compiler, JMLC. JMLC parses the annotations and generates Java code that contains assertions to validate the pre and postconditions. Thus, in a JML-based implementation of our approach (see Figure 1), JMLC can be used as the code-generator component.

The test-data generator can analyze the postconditions of an operation, and using a constraint solver, such as POOC [21], generate meaningful test data that can be returned when the operation is invoked with no precondition violations. Moreover, the test-data generator can identify equivalence classes in the output domain of an operation, and generate data to ensure coverage of each of the classes. To illustrate, consider the postconditions for the cartAdd(). By analyzing the postconditions, the test-data generator can determine that the output is either an empty string, or a non-empty string that contains two substrings, “PurchaseURL” and “Item”, with no specific ordering between them. Therefore, the generator could create three return strings: (1) an empty string; (2) a string in which “PurchaseURL” precedes “Item” (<random string>PurchaseURL<random string><random string>); and (3) a string in which “Item” precedes “PurchaseURL” (<random string><random string>PurchaseURL<random string>). Automated specification-based test-generation techniques (e.g., [7]) can be used to ensure adequate coverage of the properties specified as postconditions.

The semantic service stubs, along with test data, can be locally deployed by clients, and used, instead of the remote services, during initial testing of their applications.

### 3 Stub-Based Testing

Testing a service-oriented application requires that all interactions between the application and the invoked services are tested adequately.\(^5\) The goals of such testing include: validating the code that creates input messages sent to the invoked services; validating the code that parses output messages from the services; ensuring that all service exception conditions are processed correctly; and finally, verifying that the service provides the business logic expected by the application. Some of these goals can be accomplished by using locally deployed semantic service stubs. (In the subsequent discussion, we use $A_{RS}$ to refer to the client application integrated with remote services and $A_{LS}$ to refer to the application integrated with local semantic stubs.)

The use of semantic service stubs can reduce the costs of testing client applications and potentially lead to early detection of faults. To illustrate these benefits, we present a classification of client test cases, shown in Table 1. The classification is based on the outcome of a test case (when executed on $A_{LS}$) and whether a test case violates service contracts. A typical test suite for an application can include randomly generated test cases, code-coverage-based

### Table 1. Classification of client test cases into five categories based on (1) the outcome of a test case on $A_{LS}$ and (2) whether a test case violates service contracts.

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{fail(v)}$</td>
<td>Failing test cases with invalid service inputs that are provided by the user (reveal application fault in filtering invalid inputs or handling service exceptions)</td>
</tr>
<tr>
<td>$T_{fail(v)}$</td>
<td>Failing test cases with invalid service inputs that are generated in the application (reveal application fault in creating input messages)</td>
</tr>
<tr>
<td>$T_{fail(v)}$</td>
<td>Failing test cases with valid service inputs (reveal application fault in code that is unrelated to processing of service inputs and exceptions)</td>
</tr>
<tr>
<td>$T_{pass(v)}$</td>
<td>Passing test cases with valid service inputs (designed to test exception-handling code)</td>
</tr>
<tr>
<td>$T_{pass(v)}$</td>
<td>Passing test cases with valid service inputs</td>
</tr>
</tbody>
</table>

\(^5\)In this paper, we focus on functional testing, and do not consider other types of testing such as performance testing.
test cases, and functional test cases; we call this test suite $T_{\text{initial}}$. We partition this test suite into five categories.

The first three test-suite categories in Table 1 contain test cases that fail when executed on $A_{LS}$. First, executions of some of the test cases may fail because, in those executions, the application invokes the service with invalid inputs that are provided by the user ($T_{\text{fail}(in_u)}$). Such test cases reveal faults in the application—either the application should filter out such invalid user inputs or handle exceptions that are returned by the service in response to the invalid inputs. Second, some test cases may fail because they invoke the service with invalid inputs that are generated in the application ($T_{\text{fail}(in_a)}$). Such test cases reveal faults in the application code that creates the input messages. For example, the application may be formatting valid user inputs incorrectly before invoking the service. Third, some test cases may fail because of faults in the application code that is unrelated to the interface with the service—e.g., code unrelated to creating input messages and handling exception conditions ($T_{\text{fail}(v)}$). Test cases in this category invoke the service with valid inputs.

Semantic service stubs can enable such faults to be detected without invoking remote services. Thus, the faults can be fixed so that the application is more robust when it is integrated with remote services, which avoids discovery of the faults on $A_{RS}$ and the consequent re-execution of the failing test cases on $A_{RS}$ after the fault repair.

The fourth category of test cases ($T_{\text{pass}(in)}$) includes test cases that invoke the services with invalid inputs (either user-provided or application-generated) but that do not fail. Such test cases are designed to exercise the application’s behavior when it receives exception conditions from the service, and are essential for testing the robustness of the application. Finally, the remaining test cases ($T_{\text{pass}(v)}$) neither violate the service input constraints nor fail.

Test cases that appear in $T_{\text{fail}(in_u)}$ or $T_{\text{pass}(in)}$ need not be run on $A_{RS}$. Test cases in $T_{\text{fail}(in_a)}$ invoke the application with incorrect inputs and result in failing executions. After faults have been identified and fixed, these test cases can be rerun on $A_{LS}$ to verify the fault repair. Tests in $T_{\text{pass}(in)}$ are designed to test whether the application invokes the service incorrectly or does not handle, or incorrectly handles, an exception response from the service; such behavior can be adequately tested using stubs. In both these cases, after the faults identified by the tests have been fixed, the tests need not be run on $A_{RS}$.

Thus, one of the key benefits of our approach is that it can enable early detection of faults in the client application. Moreover, certain aspects of the application functionality can be tested adequately without invoking the remote services (and incurring the associated costs). Finally, the approach can reduce the size of the test suite that needs to be executed on $A_{RS}$.

4 Case Study

To demonstrate the feasibility of our approach and evaluate its benefits, we implemented a prototype and conducted two case studies.

4.1 Experimental setup

Services and semantic service stubs. For our study, we used a set of data services from an ongoing IBM project. There are 12 key artifacts in the system, which include business processes, requirements, and key performance indicators (KPIs). For each artifact, there are REST-based services for retrieving, creating, updating, and deleting the artifact, and linking/unlinking it to another artifact. On average, each artifact has 10 operations associated with it. The total services code consists of about 20000 lines of Java code.

One of the authors who was involved in the development of the services annotated the code with JML specifications. The specification consists of preconditions on input data and postconditions on return values. We used JMLC to generate the stubs containing assertions for validating the preconditions. Then, we manually augmented the stub code to generate return values based on postconditions of an operation (e.g., to return values of different classes, such as null and valid values, as discussed in Section 2.3).

Client application. We wrote a client Java application that invokes the services to implement five use cases: (1) create a business process; (2) update a business process; (3) retrieve a business process; (4) link KPIs to a business process; (5) unlink the KPIs associated with a business process. The client application consists of 5 classes and about 2000 lines of code; two of the classes interact with the data services. The client was written by an author who was not involved in the development of the services and, therefore, was not aware of all the correct semantics for invoking the services.

We refer to the application as $A_{bp}$ when integrated with remote services, as $A_{LS}$ when integrated with semantic service stubs, and as $A_{bp}$ when the context of integration is irrelevant.

Test suite for the client application. Next, we used a random test-case generator JCrasher [8] to generate JUnit test cases for the client application.\textsuperscript{6} JCrasher generated 47 test cases for the application. We augmented the test suite by generating 50 functional test cases.

Deployment. Finally, we deployed the service over Tomcat on a Linux server (3 GHz, 7GM RAM); we deployed the semantic stubs as a service over Tomcat on a Windows machine (2.16 GHz, 2GB RAM). The client application, along with the test cases, resided on the Windows machine too.

\textsuperscript{6}We selected JCrasher because it has been used in several empirical evaluations and is widely cited (e.g., [18, 20]).
4.2 Study 1: Test suite reduction

**Goals and method.** In the first study, we investigated the reduction in the size of the test suite that must be executed on $A_{RS}^{bp}$. Table 1 shows that test suite $T_{final}$ contains the test cases to be executed on $A_{RS}^{bp}$. Thus, the difference in the sizes of $T_{initial}$ and $T_{final}$ represents the test suite reduction that can be achieved using our approach. We executed all 97 test cases on $A_{LS}^{bp}$. For the failing test cases, we identified the ones that were caused by faults in the application. We fixed those faults and re-executed all the test cases. By following this process, we classified the 97 test cases into the categories shown in Table 1.

**Results and analysis.** Table 2 shows the classification of the 97 test cases into different categories. During the first execution of the 97 test cases, 47 passed and 50 failed. Among the passing test cases, none invoked the service operations with invalid inputs. Therefore, the 47 passing test cases belong in category $T_{pass(v)}$.

On examining some of the failing executions, we found that $A_{bp}^{bp}$ was passing invalid user inputs to the service operations. We fixed the fault by adding checks in $A_{bp}^{bp}$ that filtered out invalid inputs. Then, we reran all 97 test cases. In this run, 77 test cases passed, whereas 20 failed. Thus, the 30 additional test cases that passed after the fault repair had previously failed because invalid user inputs were being passed in to the service; we classified these 30 test cases into category $T_{fail(u_m)}$.

Next, we examined some of the failing executions in the second iteration, and found another fault in $A_{bp}^{bp}$: the application was incorrectly formatting the string containing KPIs that was input to the service operations that link and unlink KPIs. The application used a semi-colon as the delimiter in the input string, whereas the operation expected a comma as the delimiter. Upon fixing this bug, we reran the 97 test cases; in this iteration, all test cases passed. Thus, the 20 test cases that had failed in the previous iteration revealed a fault in input message creation in $A_{bp}^{bp}$. Therefore, we added these test cases to the $T_{fail(u_m)}$ category.

The final test suite, $T_{final}$, that needs to be executed on $A_{RS}^{bp}$ contained 67 test cases: 47 from $T_{pass(v)}$ and 20 from $T_{fail(u_m)}$. As discussed in Section 3, test cases in $T_{fail(u_m)}$ need not be executed on $A_{RS}^{bp}$. Thus, the size of the test suite decreased from 97 to 67—a reduction of 31%.

Although limited in nature (we discuss threats to the validity of our observations in Section 4.4), the study indicates that our approach can be effective in reducing the size of the test suite that needs to be executed on $A_{RS}^{bp}$. Moreover, the execution of test cases on $A_{LS}^{bp}$ can lead to early detection of faults, thus ensuring that certain aspects of the application functionality are well-tested before service integration occurs.

4.3 Study 2: Execution efficiency

**Goals and method.** In the second study, we evaluated the improvement in execution time of the test suite that may be achieved by using local semantic stubs. We considered two scenarios. In scenario 1, we executed the test suite using our approach. Thus, we executed $T_{initial}$ twice on $A_{LS}^{bp}$ and $T_{final}$ on $A_{RS}^{bp}$. The two executions of $T_{initial}$ simulate the situation in which faults are identified during the first run and, after fault repair, the test suite is rerun to verify the fixes. We computed the total execution time as:

$$T_{ls} = 2 \times time(T_{initial}, A_{LS}^{bp}) + \text{time}(T_{final}, A_{RS}^{bp})$$

where $\text{time}(T, A)$ represents the time required to run test suite $T$ on application $A$.

In scenario 2, we considered the conventional test execution environment in which semantic stubs are not used. To compute the test execution time for this environment, we ran $T_{initial}$ twice on $A_{RS}^{bp}$:

$$T_{rs} = 2 \times \text{time}(T_{initial}, A_{RS}^{bp})$$

Moreover, for the execution of test cases on $A_{RS}^{bp}$, we considered varying server delays to simulate potential network latency and server loads, which may affect test-suite execution in a real environment. To experiment with different server delays, we deployed a router on the Linux server that routes requests to the service after introducing a specified delay. We computed $T_{ls}$ and $T_{rs}$ for five values of server delays (in seconds): 0, 0.25, 0.5, 0.75, and 1.0.

**Results and analysis.** Figure 3 plots the execution times for the two scenarios. The horizontal axis shows the server delay values; the vertical axis shows the execution time (in seconds) for $T_{ls}$ and $T_{rs}$. The plot illustrates that the execution time for $T_{rs}$ increases significantly faster than the execution time for $T_{ls}$. At zero server delay, $T_{ls}$ is 33 seconds, whereas $T_{rs}$ is 42 seconds. At one-second delay, $T_{rs}$...
(861 seconds) is nearly three times $T_{ls}$ (278 seconds). On average, the execution time for $T_{rs}$ is more than twice the execution time for $T_{ls}$.

Thus, this study indicates that test execution using local semantic stubs can be much faster than test execution using remote services only. With larger test suites and more complex client applications, the differences between the two scenarios might be much more significant.

4.4 Discussion

The empirical results indicate that our approach can be effective in reducing the size of the test suite and improving the execution efficiency. However, there are some threats to the validity of the evaluation.

The most significant threats in our validation are threats to external validity, which arise when the results cannot be generalized. One external threat is the representativeness of our subject program and test suite. Our evaluation is based on only one application that was written in-house by the authors; therefore, we cannot make any claims about how the results might apply to more varied services and service-oriented applications. We tried to limit this threat by using services that have been developed, and are used, in an ongoing research project. We also wrote the application by creating realistic use cases. Moreover, the application was written by an author who was not involved with the development of the services.

Our test suite may not be representative of how test cases are generated in real environments. We tried to mitigate bias in test-case generation by using a combination of random and functional test cases. However, the composition of a test suite is a significant determinant of the effectiveness of our approach, which may have affected our observations.

Another threat concerns the nature of the faults detected on the application. This threat is reduced by the fact that the faults were not seeded and, in fact, were detected unexpectedly only during test execution. A final threat to external validity is our selection of the testing process that the entire test suite is rerun after a fault repair. A selective regression-testing technique would rerun only the test cases that are affected by the fault repair. However, we think that in environments where automated test execution is used, rerunning all test cases is not unrealistic.

To conclude, although our study is limited in nature, it illustrates the benefits of our approach. Future studies that are more elaborately designed can investigate whether the results generalize.

5 Related Work

There is much research in the area of web-services testing; however, most of the existing work focuses on test-data generation and developing test automation frameworks (e.g., [1, 12, 13, 18, 22, 26]).

Many researchers have presented the idea of associating contracts with web-service interfaces to support testing (e.g., [9, 10, 22, 23, 25]). Some researchers have proposed extensions to conventional contracts to enable richer behavioral specifications for web services, such as input-output dependences and invocation sequences [25], and control constructs [9]. The contracts are used to create test cases (using, for example, preconditions, control constructs, and invocation sequences) and test oracles (using postconditions) [9, 10, 22, 23]. Contracts have also been used to generate conformance tests, which validate, before registering a service, that the service is delivering the expected functionality [2, 11]. Our approach differs from these techniques in that it leverages contracts to generate semantic stubs that are used by service consumers for testing their applications. Unlike existing research, the goal of our work is to improve the testing of client applications—by enabling test suite reduction, efficient execution of test cases, and potential early detection of defects.

Pacheco and Ernst [20] present a classification of test cases as illegal, normal operation, or fault-revealing. Their approach uses an operational model of a system (consisting of properties that hold at method entry and exit) to classify a candidate input based on the properties that are violated. The goal of their approach is to create a reduced set of fault-revealing test cases from a large test suite. Our classification of test cases is more fine-grained that their classification: for example, the test inputs classified as normal operation correspond to $T_{pass,(in)}$ and $T_{pass,(u)}$ in our classification. Our classification is intended to capture how test-case properties determine whether a test case should be executed on remote services, and thus, illustrate the test suite reduction that can be attained using our approach.

Using mock objects is a well-known technique for substituting parts of a program that are irrelevant for testing a unit [17]. Tillmann and Schulte [24] present a technique for generating mock objects, that corporate behavior, by analyzing client test cases and applications that use the mocked objects. Our approach applies the notion of behavioral mock objects—in the form of semantic stubs—to the domain of service-oriented applications. In this domain, semantic stubs offer the benefits of reducing the costs of testing by enabling fewer invocations of remote web services.

6 Summary and Future Work

In this paper, we presented an approach that uses semantic service stubs for improving the efficiency and effectiveness of testing service-oriented applications. Our approach can be used to partition the client test suite into subsets such that not all subsets need to be executed using remote services. The results of our study indicate that the approach can not only reduce the number of test cases that must be executed on remote services, but also lead to early detec-
tion of faults, especially those related to the application’s interaction with the services.

In future work, semantic service annotations could be enhanced beyond preconditions, postconditions, and exception conditions to include other aspects of service behavior, such as invocation-sequence constraints and state-based constraints. To facilitate the use of our approach in practice, automated support for generating service annotations and semantic stubs is important. Thus, future work could investigate different ways of automating the process of extracting service semantics. Program-analysis techniques can be used to infer implicit contracts from source code (e.g., [4, 27]), which can help ensure that the service contracts are complete and consistent. Moreover, documentation available in HTML pages for services, such as those provided by Google and Amazon, can be parsed using screen-scraping technologies, to recover service contracts. Finally, future experimental studies, using a wider variety of services and applications, could evaluate how our results generalize.

References