A Framework for Automated and Composable Testing of Component-based Services

Miguel A. Jiménez∗, Ángela Villota Gómez∗, Norha M. Villegas†, Gabriel Tamura∗ and Laurence Duchien†

∗ Department of ICT, Icesi University, Colombia - Email: {majimenez, apvillota, nvillega, gtamura}@icesi.edu.co
† LIFL – University of Lille 1 CNRS and INRIA Lille, France - Email: laurence.duchien@inria.fr

Abstract—The vision of service-oriented computing has been largely developed on the fundamental principle of building systems by composing and orchestrating services in their control flow. Nowadays, software development is notably influenced by service-oriented architectures (SOAs), in which the quality of software systems is determined by the quality of the involved services and their actual composition. Despite the efforts on improving their individual quality, adding or replacing services in an evolving system can introduce failures, thus compromising the satisfaction of the system’s functional and extra-functional requirements. These failures erode the trust in the SOA vision. Thus, a key issue for the industrial adoption of SOA is providing service providers, integrators, and consumers the means to build confidence that services behave according to the contracted quality conditions. In this paper we present a first version of PASCANI, a framework for specifying and executing test specifications for service-oriented systems. From a test specification, PASCANI generates a configuration of testing services compliant with the Service Component Architecture (SCA) specification, which can be composed to integrate different testing strategies, being these tests traceable in an automated way. Our evaluation results show the applicability of the framework and a substantial gain in the tester’s effort for developing tests.

I. INTRODUCTION

In the service-oriented computing vision, software development is based on service provisioning, orchestration and composition. This composition of loosely coupled services allows to create flexible, dynamic business processes and applications that trespass the boundaries of organizations, involving third-party computing platforms. Therefore, the quality of such systems depends on the quality of all of these involved services and their actual composition and orchestration. Moreover, these systems also evolve, either because some of their services change (e.g., by versions) or because other services with similar, but improved, functionalities appear, or because of new system requirements. Given that services are replaceable by definition, this evolution is performed by adding or replacing them in the service composition chains. Nonetheless, even though services are usually tested individually in their sources of origin (e.g., third-party providers), adding or replacing them can introduce unit, integration, or system level failures, thus compromising the satisfaction of functional and extra-functional requirements (i.e., the system quality).

To maintain service-oriented systems’ quality, testers develop, compose, and execute service tests in the required deployment configurations. This process is error prone, and time and effort consuming for several factors: (i) the tester must analyze by hand the services to test, before specifying the actual tests in the service implementation programming languages, being these languages possibly different and not tailored for testing; (ii) the number of service assembly configurations valid to achieve the system goals can be high (e.g., when implementing dynamic binding), and so the respective testing configurations; and (iii) ideally, the testing services should be reusable, composable in larger testing artifacts, and traceable by themselves.

In this paper we present PASCANI, a framework that assists testers in the specification and execution of automated, composable, and traceable tests, which are compliant with the Service Component Architecture (SCA) specification [1], for service-oriented systems. Our framework comprises two subsystems. The first is the PASCANI analyzer, a mechanism for analyzing the possible service configurations to test, which generates a test specification template from a set of selected components and services. The second is the PASCANI language and its implementation, a domain specific language for specifying tests in test modules. A test module specifies the components and services to test, the test cases for verifying the system’s functional and extra-functional requirements, and a test strategy, which specifies how to execute the test cases. For the composability, the language allows to import other test modules and compose them in the test strategy through a set of composition operators. From a test module specification, PASCANI generates a configuration of testing components, and deploys and executes it by using FRASCATI [2], an execution middleware for SCA-compliant systems.

To evaluate our approach, we integrate PASCANI into a service-oriented online shop retailer (OSR) application. Our results show the applicability of the framework by generating, deploying and executing configurations of composable tests for service provider’s services and their compositions. Our evaluation also shows time and effort gains in test development.

This paper is organized as follows. Section II presents the challenges in testing service-oriented systems that motivate this paper; Sections III and IV present the PASCANI framework, and how it supports test composition; Section V presents the evaluation of our framework; Section VI discusses related work; and Section VII concludes the paper.

II. MOTIVATION

In this section we analyze selected challenges for testing software services, and introduce our case study.

A. Revisiting Testing Challenges in Software Services

Even though it is well known that testing is a limited technique for verifying the correctness of a software system
The first challenge implied by the SOA/SCA vision is the need for testing components or services that can be composed and reused effectively. In effect, in this vision software development is based on service provision, discovery and composition in a worldwide context. Thus, as software systems involves the composition of services from third parties, it should be reasonable to expect that corresponding testing services, not even with instructions for testing them. On the one hand, for service providers test development is costly: first, because its engineering effort, and second, because the execution of functional and extra-functional tests in their infrastructures can degrade quality attributes of the provided services themselves. On the other hand, for service integrators it is reasonable to ask for minimum guarantees on the quality of the services to integrate and compose, at both functional and extra-functional levels. Given the benefits and despite its limitations, testing is widely used in practice to provide confidence in software services quality [3], [4], [5].

In the next sections, we present the specific testing challenges addressed in this paper. It is worth noting that, even though we target service testing, our approach follows the vision of the service-oriented architecture (SOA) as based on the service-component architecture (SCA) [6], [1]. That is, we realize services through software components and corresponding middleware, which ensure interoperability with services implemented in common technologies and exposed through well-known protocols.

1) **Composability:** The first challenge implied by the SOA/SCA vision is the need for testing components or services that can be composed and reused effectively. In effect, in this vision software development is based on service provision, discovery and composition in a worldwide context. Thus, as software systems involves the composition of services from third parties, it should be reasonable to expect that corresponding testing artifacts (i.e., software components) should also be composable in order to maximize their cost-effectiveness, thus leveraging their testing value beyond their original goals. Then, “service-based system tests” for ensuring requirements satisfaction could be seen as a hierarchy of reusable and composable tests, starting from functional ones at the unit (i.e., individual service) level, and building up through layers of integration tests (i.e., for composed services), and finally, a set of extra-functional tests (i.e., at the system level, for instance measuring the performance of the resulting combined execution of own and third-party integrated services).

Furthermore, this challenge is exacerbated by the extra-mural nature of the services to compose, given the lack of control over the respective testing provisions. That is, service integrators require tools that assist them in the specification and implementation of testing artifacts. Naturally, an adequate testing language with precise semantics is required for expressing not only testing functionalities but also, and more importantly, how to compose these functionalities and artifacts, at different levels of application. For instance, for providers, services are endpoints of software units without specific contexts of use, and thus, they are mainly concerned about unit tests. Unfortunately, they usually provide only PDF instructions to perform manual tests, at best. A service integrator, on the other side, usually must develop not only unit tests for these services, but also integration and system tests, for having minimum guarantees on the whole system’s requirements satisfaction in the contexts of operation of his interest.

2) **Traceability and Logging:** A second challenge for testing services is the development of controllable logging mechanisms and standardized trace formats, applicable both to individual tests and to compositions of them, and deployable with testing artifacts automatically.

Service providers and integrators expect to deliver and compose services enabled with control and monitoring functionalities over their properties and behavior. Similarly, in service testing, traceability and logging are crucial mechanisms to provide precise information to diagnose and identify the root cause of anomalies. On the one hand, service traceability refers to the capability of tracking the states of services. From the perspective of service testing, traceability should produce trace information about two aspects: sub-test and service invocations in a test, and respective partial and accomplished test results according to the test objective. Based on this information, human testers can detect faulty services or services that are contributing the most to satisfy or dissatisfy given requirements, such as quality attributes. On the other hand, logging complements traceability by recording sensitive information about the quality of services, which is expected to be registered consistently from service and test execution. This information should be gathered dynamically from the execution of test cases and corresponding monitor probes in the target system. Traceability in service testing comprises two dimensions: behavior traceability, which refers to the degree a service facilitates the tracking of its internal and external behaviors; and trace controllability, which concerns the customization of tracking and logging functionalities [7]. Moreover, software developers usually focus on tracking the internal behavior of services rather than the external ones. As a result, services are generally better instrumented to support logging in unit tests than in integration and system tests, despite logging external behaviors are more relevant in distributed environments such as in services.

3) **Test Specification and Expressiveness:** The third challenge is the definition of adequate languages for specifying tests, with appropriate syntax and semantics, power of expressiveness, efficiency, and level of abstraction. Languages with these characteristics would allow testers to construct better tests, easier to understand, reuse, compose, trace, and maintain.

Traditionally, software developers implement test artifacts using ad-hoc approaches that are based on technology-dependent test management tools or frameworks [8]. Nevertheless, this way of implementing tests leads to unmanageable sets of testing artifacts written in different general-purpose programming languages, that is, not tailored for specifying tests. Moreover, services are usually published through interface definition languages (IDLs) such as WSDL, which testers must analyze for specifying the testing operations,
includes the translation of methods and parameters between the IDL protocols, and the interactions (composition) with other services, a process prone to errors. Thus, more than implementing tests assisted by technology-dependent tools, SOA/SCA requires domain specific languages (DSLs) that allow to specify testing components in more manageable, standardized ways. The characteristics of such DSLs for specifying tests easier to understand, reuse, compose, trace, and maintain should be [9]:

Expressiveness: DSLs must be expressive enough to capture all the necessary concepts in the domain (i.e., service testing in this case). That is, it must semantically support the full generation of test concepts and required component configurations and strategies, automatically.

Economy: DSLs must provide the user with appropriate syntax to efficiently express sentences of their domain, and to encapsulate and reuse language modules.

Adequate level of abstraction: DSLs must be effective, that is, the notations and abstraction level must be as close to the domain-experts language as possible. For instance, developers are more familiar with programming languages than with formal specification languages. Nonetheless, neither of these two kinds of languages (programming and formal) are well suited specifically for testing.

B. A Running Example

The following case study is used along this paper to explain the application of our proposed testing framework. In electronic commerce, an OSR application allows customers/users to shop for products and services through the Internet. For this, users browse a catalog of products and select the ones to be purchased by adding them to a shopping cart. Furthermore, with the proliferation of personal context and user-centric recommendations, users expect from OSR applications product recommendations based on their preferences and situations. To complete the purchase, users initiate the checkout process by entering information about delivery and payment preferences. Delivery preferences include information such as the shipping address and the delivery service preferred by the user. Payment information includes data about the preferred payment method such as the credit card number, security code, holder name and billing address. To recommend relevant products, complete the checkout process, and confirm the order delivery, OSR applications composite several services provided by third party applications. In particular, for recommending relevant products, OSRs may implement a service based on collaborative filtering techniques, which in turn can consume third party services to perform computationally expensive operations, such as matrix multiplication [10]. Similarly, for completing the checkout process, OSR applications typically use a postal address verifier (PAV) service to verify that the shipping and billing addresses do exist; a credit card number verifier (CCNV) service to confirm all the information about credit cards; and a delivery service (DS), with its corresponding delivery tracking service (DTS), to dispatch the order and for the user to monitor the delivery process. Several instances of these services are already run on the internet and available to be used, both manually by humans and programmatically through wsdl interfaces.

III. THE PASCANI FRAMEWORK

PASCANI, as a framework for generating and executing testing services, is based on the Service Component Architecture (SCA) specification. SCA is a programming model conceived to serve as a foundation for realizing SOA systems. SCA provides specifications to create and compose components providing services to build complete applications. Components can rely on different environments and technologies, given that SCA is independent of implementation and communication mechanisms [1]. Figure 1 depicts a partial view of the SCA architecture for our running example, including three of the OSR components corresponding to the functionalities described in Sect. II-B, namely Purchase, Recommender, and Delivery. The legend in the figure depicts the main SCA notational elements: Components implement the functional parts of the system and can be combined into larger structures called Composites. Components interact with others through Services via provided or required interfaces, connected with a line called Interface Binding.

![SCA diagram for the Online Shop Retailer example.](image)

Figure 1: SCA diagram for the Online Shop Retailer example.

PASCANI comprises two subsystems: (i) the component and service analyzer, which generates a test specification template from a set of selected components and services; and (ii) the testing specification language, which includes a translation model (i.e., a compiler) and an execution model. That is, the framework assists users (i.e., human testers) in both describing the system to be tested in terms of its components and services; and specifying test modules with the corresponding test cases and strategies for verifying the system’s functional and extra-functional requirements.\(^\text{1}\)

In this section we focus on the language definition that allows the specification of test modules, from the perspective of the service provider. To better explain this, we elaborate on the scenario of a third-party providing a service for matrix multiplication, as described in our running example (cf. Sect. II-B). Assume the service provider uses the Strassen’s algorithm for implementing its matrix chain multiplication service.

\(^1\)The current version of PaSCANI is available from the SVN repository http://pascani.org upon request.
(cf. MCM in Fig. 1), and uses PASCANI for the specification and composition of its tests.

### A. Language Specification

The main unit for specifying tests in PASCANI is the *test module*, and it defines two types of them: *basic* and *composed*. A basic test module is never composed of other test modules and is intended only for two testing targets: unit and system testing. In contrast to basic modules, composed test modules reuse other test modules, and may have different testing targets. A test module specifies the system’s components and services to be tested, as well as the test cases with their corresponding execution control flow. We also refer to this control flow as the *test strategy* of the module. Modules are the mechanism to both specify a deployable system and its test components, specify test strategies, and compose tests. A PASCANI module is defined by two mandatory blocks, *system structure* and *tests*, and an optional block named *control*. Control blocks are similar to main methods in programming languages such as Java and C++, thus they are optional in modules that do not specify a test execution strategy.

Listing 1 illustrates the structure of a basic PASCANI module for testing the implementation correctness of the classical Strassen’s algorithm that is used in our case example (cf. Section II-B).

```java
1 package org.strassen;
2 import java.util.Arrays;
3 import org.strassen.Matrix;
4 module StrassenTest {
5    composite strassen = loadComposite(
6        "strassen.composite"
7    ) providing services {
8        rmi mcm:rmiservice@"remote01":1099
9        with interface org.strassen.MatrixService
10    } testcase checkCorrectness {
11        int[][] A, int[][] B, int[][][] exp
12    } using strassen.mcm {
13        // Multiply A times B and verify if answer
14        // is equal to the expected matrix
15        int[][] C = strassen.mcm.multiply(A, B);
16        test isCorrect = Arrays.deepEquals(C, exp);
17        labeled "Multiplication correctness"
18        message when
19        passed:"The algorithm was tested successfully"
20        failed:"Wrong implementation"
21        return isCorrect;
22    } control {
23        int[][] A = Util.getImageData("/tmp/A.jpg");
24        int[][] B = Util.getImageData("/tmp/B.jpg");
25        int[][] C = Util.getImageData("/tmp/C.jpg");
26        checkCorrectness(A, B, C);
27    }
28 }
```

Listing 1: A test module for Strassen’s algorithm.

**System Structure Block.** This mandatory block specifies the architectural configuration of the system to be tested (cf. lines 5–10 in Listing 1). It defines variables used to represent the corresponding components (cf. line 5) as well as services and interfaces (cf. lines 7–10). The specification of the system to test can include two types of components: anonymous and known components. *Anonymous* components are those that have been already deployed (e.g., by a third-party provider somewhere in the Internet) and must be specified using the reserved word *anonymous*. *Known* components are existing

SCA components and must be deployed by PASCANI before executing the tests. The structure block in Listing 1 specifies a system to test with one known component named `strassen` that provides an RMI service `mcm`.

**Test Block.** This mandatory block allows the definition of one or several test cases using the reserved word *testcase*. A test case is a sequence of instructions which can use the services provided by the components specified in the system structure block. The result of executing these instructions must evaluate to a test value called a verdict. Verdict values are in the set $V = \{\text{passed, failed, inconclusive, exception}\}$. Lines 11–23 in Listing 1 specify the test case named `correctness` to check whether the service `strassen.mcm` multiplies a pair of matrices as expected, and returns the verdict through the variable `isCorrect` (cf. line 17).

**Control Block.** This optional block allows the definition of test strategies by specifying (i) the execution order of test case invocations; and (ii) the way of operating or composing their verdicts to obtain the final result. Control blocks may contain programming-language specific instructions (e.g., Java statements) to prepare the test environment (e.g., load test data in lines 25–27 of Listing 1).

### B. Translation Model

PASCANI translates test module specifications into SCA component configurations and corresponding programming language implementations (i.e., Java for this version of the framework). To deploy and execute the generated components, PASCANI provides an user interface and uses the FRASCATI middleware [2].

From the three blocks structure of a test module, PASCANI generates the corresponding SCA configuration as follows. The services specified in the system structure block are translated into SCA references that are bound to testcase and control components in order to be tested (cf. service `mcm` in component `Matrix` in Fig. 2 is bound to `StrassenTest_Group.StrassenTest_checkCorrectness_strassen_mcm`); each of the test cases and the control block are translated into components (i.e., `StrassenTest_checkCorrectness` and `StrassenTest_Control`) and encapsulated in the same composite (cf. `StrassenTest_Group`). Every generated component is equipped with two standard services: `run`, that is used for starting the execution of the component, and `control`, for controlling the logging functionalities (e.g., level of detail of messages to log). In our example, component `StrassenTest_Control` has references to all of the testcase components within `StrassenTest_Group`, given that it must realize the test strategy as well as the control interface for the logging and traceability of test cases execution. Finally, component `Logger` records log messages, and component `Runner` allows the tester user to start the test module execution `Runner.run`.

### C. Execution Model

The execution model of PASCANI relies on the standard SCA interfaces, services, and references defined for realizing the running, composing, logging, and tracing functionalities in
each component, as defined in the translation model. The execution model defines the way a test strategy is executed through the behavior of the generated components and services.

The sequence diagram presented in Fig. 3 depicts the execution model of the StrassenTest module of Listing 1. For simplicity, this sequence diagram focuses on the interactions among the components that realize the test strategy, and omits the ones associated with logging and control mechanisms.

The execution model steps for testing the Strassen’s algorithm implementation (cf. component Matrix in Fig. 2) is as follows. A TestClient starts the execution of the test module by invoking service Runner.run, which in turn invokes service StrassenTest_Control.run to initiate the execution of the test control. The control starts the execution of each test case, as specified in the control block of the test module, by calling the corresponding run methods. Each test case component invokes the services to be tested. In the case of the StrassenTest module, the test strategy is composed of only one test case implemented in component StrassenTest_checkCorrectness. To perform the actual test, StrassenTest_checkCorrectness consumes service Matrix.mcm, obtains the result, and compares it with the specified one to return the control block verdict. Finally, the test ends by generating the test report.

The execution model of PASCANI also includes the monitoring of properties and behavior of testing components. PASCANI components expose services to control their logging mechanism. Furthermore, test components can be monitored by reading the generated log, which constitutes the base for test traceability analysis. By default, PASCANI generates log registries at the most detailed level, that is for all of the test cases execution. However, users can turn off the logging functionalities for some or all of the components in a test module through the service Runner.control. As shown in Fig. 2, logging in PASCANI is performed through the component Logger, thus centralizing all message logs.

D. Architecture and Implementation

Figure 4 depicts the three-layer architecture of PASCANI. The lower layer constitutes the FrASCATI Runtime, an open SCA middleware that allows the dynamic deployment and re-configuration of PASCANI components at runtime. The middle layer represents the structure of the main components of the framework that were explained in previous sections. The upper layer corresponds to the PASCANI Compiler, which was implemented using the JLex 1.2.6 lexical analyzer and the CUP 0.10k parser generator. Code generation in our compiler relies on the Google closure template tool to generate the SCA testing components and their Java implementations, as well as test reports. The upper part of the diagram represents the application to be tested, including the testing components generated by PASCANI.

IV. COMPOSING TESTS WITH PASCANI

From the testing perspective, the evaluation of a software application’s quality depends not only on the results obtained from the execution of test cases, but also on the quality of the tests themselves [11]. On the one hand, several testing techniques exist for evaluating the fulfillment of functional and extra-functional aspects of quality, such as regression testing and performance testing, respectively. On the other hand, the quality of a set of test modules is usually evaluated...
against selected test criteria, for instance by measuring the coverage (i.e., percentage) of code that is executed when the test modules are applied.

Even though the test module is the basic testing unit of reuse in PASCA\textsuperscript{NI}, the key factor for achieving composability and traceability of test specifications resides in its three-blocks structure. More concretely, the reusability of PASCA\textsuperscript{NI} testing modules is achieved by three main characteristics: (i) integration of programming language statements within the control block test definition; (ii) modularity of tests; and (iii) composability of deployable testing components and the exposure of services for controlling test traceability and logging.

This section illustrates the principles used in the design of the PASCA\textsuperscript{NI} language for defining the composition operators used to realize both testing strategies and tests quality evaluation. These operators compose testing components not only generated by PASCA\textsuperscript{NI}, but also other testing components from third parties, as long as they either conform to the SCA specification, or are exposed as testing services. Moreover, to support the composition of tests results, PASCA\textsuperscript{NI} defines the operators \texttt{and}, \texttt{or}, \texttt{xor}, and \texttt{not}, associated with its verdict primitive type. Thus, test results in PASCA\textsuperscript{NI} are operated similarly as boolean expressions in programming languages, whereas test modules are composed with richer semantics, as follows.

### A. Regression Testing

This testing technique re-tests a system or component to verify that performed modifications in its code did not introduce unintended effects. In other words, after being changed, software artifacts that previously passed known tests should still pass them, except of course when the related requirements also change. Regression testing is applicable at the unit, integration, and system levels to maintain the achieved quality of an existing software system in both functional and extra-functional aspects.

In PASCA\textsuperscript{NI}, previously developed test modules $TM_1, \ldots, TM_N$ of any kind can be reused and composed for regression testing through the regression test operator, formally denoted as $\varpi$. In the language, the regression composition is expressed as $\text{regression}\{ TM_1, \ldots, TM_N \}$. The test modules are executed in sequence, avoiding repetitions in them and in service execution.

**Definition 1:** Given a set of test modules $\{TM_1, \ldots, TM_N\}$, the regression test that composes them, denoted as $TM_1 \varpi \ldots \varpi TM_N$ and abbreviated $\bigcup_{i=1}^{N} TM_i$, is defined as:

$$\bigcup_{i=1}^{N} TM_i = \bigwedge_{i=1}^{N} TM_i,\text{control()}$$

That is, the result of the regression test composition is the \texttt{and} of the verdicts returned by the test control blocks of the corresponding test modules. Nonetheless, it is worth noting that the semantics of the regression-testing composition operator, in addition to the semantics of the verdict \texttt{and} operator, also implies the automatic redirection of logging messages and traceability information to a unified \texttt{Logger} component in the generated configuration. At runtime, the execution of the test returns the verdict, and as a side-effect, it produces a record of the verdict results of the test modules included in the composition.

### B. Performance Testing

The goal of performance testing is to verify whether the system meets specified performance requirements such as response time, latency, and throughput. This testing technique applies at the unit, integration, and system levels.

Performance tests can be executed either by measuring the execution times of all the components separately and computing the overall performance by adding them, or by measuring the overall execution time at the outer scope of the service invocation. The problem with the latter is that the user may need to have a more detailed analysis of the components individual performance and in this case must include manually the performance “probing” instructions as part of the test strategy specification, thus leaving out of consideration other performance-related aspects such as networked communications time.

In PASCA\textsuperscript{NI}, previously developed performance test modules $TM_1, \ldots, TM_N$ can be composed not only for testing the overall performance of a whole system but also for profiling the services that are contributing the most to performance problems. Both functionalities are realized through the performance-test operator, formally denoted as $\ast$. In the language, the performance composition is expressed as $\text{performance}\{ TM_1, \ldots, TM_N \}$. Although the test modules are executed sequentially, there can be an alternative semantics executing them in concurrent threads. However, as the results are both summed up and also reported by test module, and the concurrent execution depends a lot on the deployment configuration using more or less machines, we found not convincing arguments for implementing this second interpretation.

**Definition 2:** Given a set of test modules $TS = \{TM_1, \ldots, TM_N\}$, the composed performance test of $TS$, denoted as $TM_1 \ast \ldots \ast TM_N$ and abbreviated $\prod_{i=1}^{N} TM_i$, is defined as:

$$\prod_{i=1}^{N} TM_i = \sum_{i=1}^{N} \text{Pascani.elapsedTime}(TM_i)$$

To realize this definition, the PASCA\textsuperscript{NI} framework associates a set of attributes to every user-defined test module. These attributes are referred through the meta-variable \texttt{Pascani} (i.e., \texttt{Pascani.elapsedTime}(TM$_i$) refers to the elapsed time of the test module TM$_i$).

The semantics of the performance-testing composition operator, besides accumulating the elapsed times of the corresponding tested services, activates the recording of their elapsed times of the executed services in the generated configuration at compile time. For having more confident results, each test module is executed a number of times that can be configured in the framework, and averaged upon finishing. At runtime, the execution of the composed test returns the total elapsed time, and as a side-effect, it produces the partial elapsed times of the involved services for which a performance test module was included in the composition.
C. Test Coverage

The third form of composition supported by PASCANI targets test coverage. As mentioned previously, test coverage is used in classical testing for evaluating the thoroughness of a set of test artifacts, measured as the percentage of code whose execution is run-through by the test suites execution. More concretely, test coverage uses a control flow-based coverage criteria whose goal is to execute every statement in a program, or specified blocks of it. It is worth mentioning that several approaches for automating the generation of test data-sets have been proposed to achieve an exhaustive coverage of all control flow paths (e.g., [12], [13]) even though it is known that this is unfeasible in general. In any case, automated generation of these data sets is outside the scope of this paper, as it addresses another dimensions of the testing problem.

In PASCANI, test coverage is evaluated as the thoroughness of a set of test modules, measured as the percentage of the services that are actually tested, with respect to the total number of provided services in all components subject to test (i.e., imported in the system structure block of the PASCANI test module specification). The test modules are executed in sequence, avoiding repetitions in them and in service execution. The composition of several test modules $T M_1, \ldots, T M_N$ for measuring their coverage is realized through the definition of the test-coverage operator, denoted as $\odot$, and expressed in the language as

\[
\text{coverage}\{ T M_1, \ldots, T M_N \}.
\]

Definition 3: Given a set of test modules $T S = \{ T M_1, \ldots, T M_N \}$, the test coverage of $T S$, denoted as $T M_1 \odot \ldots \odot T M_N$ and abbreviated $\bigodot_{i=1}^{N} T M_i$, is defined as:

\[
\bigodot_{i=1}^{N} T M_i = \frac{\|\{s \mid \forall c \in \text{Pascani}.T M_i.\text{Components} \ (s \in c.\text{providedServices} \wedge \text{s in Pascani.Services.markedAsExecuted})\|}{\|\{s \mid \forall c \in \text{Pascani}.T M_i.\text{Components} \ (s \in c.\text{providedServices})\|}
\]

where $\{s \mid \forall c \in \text{Pascani}.T M_i.\text{Components} \ (s \in c.\text{providedServices})\}$ is the defined-by-comprehension set of the provided services $s$ that are defined in all of the components imported in test module $T M_i$; and $|X|$ is the cardinality of the set $X = (\text{count(testedServices)} / \text{count(totalServices)}) * 100\%$.

That is, as with the performance-testing composition operator, PASCANI includes a record of the executed services in the attributes associated to every test module $T M_i$, which is updated dynamically at runtime as the services involved in the test modules are executed. In this definition, Pascani.TM$_i$.Components refers to all of the imported components in the system structure block of $T M_i$, whereas Pascani.Services.markedAsExecuted refers to the services actually executed in the given execution.

Thus, the semantics of the test-coverage composition operator not only computes the ratio between the number of actually tested services and the total number of them (expressed as a fraction of 100%), but also activates the recording of the executed services in the generated configuration. At runtime, the tests execution returns the percentage of test coverage, and as a side-effect, produces the record of the services that were actually tested (in cascade) from these tests execution. The total number of defined services is obtained through the PASCANI analyzer. Both functionalities, the update of test module attributes and the count of the total number of defined services in the set of imported components, are supported through an extension of the FRASCATI SCA middleware and its introspection capabilities developed by Tamura et al. [14].

V. Evaluation

This section presents an evaluation of two aspects of PASCANI: its practical feasibility for composing testing services, and the relative gain it offers in development effort for specifying testing artifacts. To conduct this evaluation, we analyze the application of PASCANI in the service provider side using the matrix-chain multiplication (MCM) problem. Even though we also applied PASCANI for the service integrator side in the OSR application, we omit details of the respective results due to space constraints.

A. Composing Tests: the Service Provider Side

Assume the MCM service provider splits the matrix-chain multiplication problem into three different subproblems: (i) the matrix-pair multiplication problem; (ii) the matrix-chain parenthesesization problem, which finds the multiplication order that minimizes the number of scalar multiplications; and (iii) the matrix-subchain multiplication scheduling problem, which takes the output from (ii) and finds subsets of matrix multiplications that can be executed concurrently to decrease the total multiplication time [15]. Fig. 5 depicts the variability of the MCM problem in the form of a feature model.\(^2\) The solutions for each of these three subproblems are represented as features MatrixPairMultiplier, Parenthesizer, and Scheduler, respectively (cf. second level in Fig. 5). Moreover, each feature in the model is associated with an SCA component.

\[\text{Figure 5: Feature model for the matrix-chain multiplication solution.}\]

To realize the possible configurations of the MCM service, we implemented seven SCA components. For solving the matrix-pair multiplication problem we implemented three alternative components: Strassen, BlockReduce, and Hybrid. Strassen implements the sequential solution proposed by Volker Strassen, BlockReduce implements a map-reduce strategy, and Hybrid implements a variation of the block-reduce strategy. For solving the matrix-chain parenthesesization and the matrix-subchain problems, we implemented two alternative components for each case: Sequential and Distributed.

\[\text{2This model is available in the SPLVT tool: } \text{http://goo.gl/6DUUR5}\]
The fundamental motivation behind PASCANI is the composition and reuse of test modules, which is achieved through the specification of unit, integration, and system test modules. PASCANI allows the specification of integration and system tests by invoking the test cases of imported modules, and the composition of test modules through the regression ($\circ$), performance ($\oplus$), and coverage ($\ominus$) operators presented in Section IV. In light of this, each component of the MCM solution is equipped with one or several basic test modules that address different test objectives. We implemented basic test modules for testing the correctness and performance of several of the SCA components used in the MCM problem of our case study. For example, component Strassen is accompanied by a basic test module $TM_{StrassenTest}$ that tests its correctness. The target of these two modules is unit testing, in its functional and extra-functional aspects, respectively.

In this case study, PASCANI supports the service provider in the implementation and execution of the integration and system tests required to validate the correctness and performance of each of the fifteen MCM products that can be obtained from the variability model presented in Fig. 5. The explanations in this section use two of these possible products: $MCM_1$ and $MCM_2$. Assume that product $MCM_1$ is defined by component MatrixPairMultiplier.Strassen (also referred as Strassen) for the sake of simplicity, whereas product $MCM_2$ by components Parenthesizer.Sequential and Scheduler.Sequential. The tester must specify test modules that call the test control blocks or selected test cases of the modules associated with the SCA components that define each product configuration. For example, for testing the correctness of product $MCM_2$, the tester must specify an integration module $TM_{IntegrationTest}$ whose execution produces a verdict on the integration. This verdict results from the operation applied to the values returned by the basic test modules in charge of testing the individual correctness of its three components.

1) Regression Testing: Suppose now that the implementation of component Strassen is modified. Because of this change, which does not affect the component service interfaces, the tester must implement a regression test to guarantee that product $MCM_2$ still satisfies its functional or extra-functional requirements. For this, the tester must create a composed test module $TM_{RegressionTest}$ that specifies in its control block a regression test that composes the unit test of component Strassen with the integration test of product $MCM_2$ using the regression operator: $TM_{StrassenTest} \oplus TM_{IntegrationTest}$. That is, if the Strassen component implementation is changed (because e.g. of a new version or an evolutionary change of the algorithm base), it is necessary to retest that both the unit tests of the component correctness and its integration test with the Parenthesizer and Scheduler components are still successfully passed. Fig. 6 depicts the SCA (partial) configuration for the regression test.

A regression test report in PASCANI typically includes, for each test module, its total execution time; a flag to indicate whether the module was executed in regression test mode; and for each of the services tested by its test components, a general flag to indicate whether the service was tested, and the execution time and verdict for each test case associated with the tested service.

2) Performance Testing: To test the performance of product $MCM_2$, the tester must compose the primitive modules that test the performance of components Strassen, Parenthesizer.Sequential, and Scheduler.Sequential: $TM_{StrassenTest} \ominus TM_{Paren Patriots} \ominus TM_{SchedulerSequential} \leq Target_{maxLimit}$. From this composed module, PASCANI generates the corresponding SCA configuration. The performance reports, not shown in this paper due to space constraints, contains the general verdict and performance measures obtained for product $MCM_2$, as well as the verdicts and measures for each individual component. This level of detail in the logging of test results is possible thanks to the composition semantics of the performance tests operator, and the tracing and logging facilities of PASCANI.

Table I shows the outputs obtained when using the performance composition operator to test products $MCM_1$ and $MCM_2$ for the multiplication of matrices of sizes $640 \times 480$, $480 \times 320$, $320 \times 240$, and $240 \times 64$ among others.

<table>
<thead>
<tr>
<th>Component</th>
<th>$MCM_1$ (ms)</th>
<th>$MCM_2$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parenthesizer</td>
<td>–</td>
<td>1</td>
</tr>
<tr>
<td>Scheduler</td>
<td>–</td>
<td>1</td>
</tr>
<tr>
<td>Strassen</td>
<td>3447</td>
<td>4774</td>
</tr>
<tr>
<td>Total time</td>
<td>3449</td>
<td>4774</td>
</tr>
</tbody>
</table>

Table I: Performance tests results for products $MCM_1$ and $MCM_2$. 

![SCA (partial) configuration for the regression test.](image)
3) Test Coverage: The composition of coverage tests is specified similarly as the regression and performance tests composition: a module must be created, using the coverage operator (⊙) to compose the test modules for which the measure is required. For example, the coverage achieved when composing the basic test modules associated to MCM2: TMStrassenTest ⊗ TMParenthesizerTest ⊗ TMSchedulerTest corresponds to 100%, because the composition has three services to test: Scheduler.start_multiplication, Parenthesizer.parenthesize, and Strassen.mcm, and each service is tested by the corresponding basic test module. However, as the complexity and number of component tests grow, the more difficult is to determine the percent of coverage manually.

B. Engineering Effort

Table II presents the relative gain of developer’s effort when specifying tests with PASCANI in our case study. The results show an effort gain superior to 70% for all cases. This is mainly because PASCANI releases the developer from implementing functionalities of the SCA layer, such as protocol translation and message marshaling and unmarshaling.

In this table, column Component refers to the service components we tested. For evaluation purposes, each test case was defined in a separate test module (cf. column Test Module) thus leaving the measurement at the coarsest level of granularity. Column PASCANI LOCs refers to the number of lines of code written in PASCANI by the developer. Column Generated LOCs includes two sub-columns: Project, which refers to the classes that are automatically generated in correspondence with the test modules; XML, which refers to the generated SCA composites. Finally, column Effort refers to the developer’s effort when writing tests with PASCANI, and column Reduction the relative effort gain obtained when using PASCANI.

VI. RELATED WORK

This section presents related work, relevant to service-oriented systems, in light of the categories of challenges discussed in Section II-A: composability, traceability and logging, test specification and expressiveness.

Concerning composability, we found no approaches that support the composition of testing artifacts for testing software services. Component compositions may have an important number of variation sources, which increase exponentially with the number of individual testing component options. Therefore, the composition of existing unit tests can help reduce the costs of the testing process.

The lack of composable and reusable testing artifacts limits the effectiveness of the validation process in service-oriented systems [7], [4]. This is in part because of the extramural nature of services that makes impractical to anticipate all the possible deployment configurations of the composition, thus test cases cannot be executed in the real context of execution. Moreover, systems implement adaptive behaviors through dynamic binding, either by replacing individual services or adding new ones, and thus integration testing has to deal with changing configurations that cannot be anticipated [16]. To counteract this issue, several approaches rely on simulation and prediction mechanisms to mimic third party components and evaluate their behavior and properties [17], [18]. PASCANI provides suitable mechanisms not only to instrument services with testing services, but also to specify and implement test cases that can easily compose these testing services at any time according to changes in service compositions.

Regarding traceability and logging, the lack of instrumentation to support these functions in service testing augments the complexity of understanding component behaviors and reporting errors. We found no evidence of approaches to trace tests at every level. Moreover, we found that logging in service testing is better instrumented for unit tests than for integration or system tests. Therefore, keeping track of external behaviours is still an open challenge for distributed environments. PASCANI contributes to this issue by providing automatically deployable logging mechanisms applicable to individual tests and their compositions. PASCANI also allows software testers to select the types of traces applicable in a particular situation, or decide on the level of logging (e.g., by enabling logging functionalities for a selected set of test cases).

Finally, with respect to test specification and expressiveness, we found only one domain specific language (DSL), based on ontologies [19], for the specification of tests in service-oriented applications. However, we found no DSLs suitable for the specification of reusable, composable, and traceable service-oriented tests. In general, the focus is, rather than on specifying tests themselves, on generating test cases from service specifications such as WSDLs [17], [20], or from UML or BPEL models, a common approach in the model-based testing community [21]. PASCANI defines composition operators whose semantics guarantee that the generated testing components automatically have these characteristics. It is worth mentioning that most of the PASCANI language testing concepts were borrowed from TTCN-3 (cf. http://www.ttcn-3.org). Despite TTCN-3 is a standardized language and provides a full stack for testing software at any stage, its syntax and semantics remain very close to low-level communication testing, therefore extending it for business application developers was not a suitable option.

VII. CONCLUSION AND FUTURE WORK

In this paper we presented PASCANI, a framework for specifying, and executing automated, composable, and traceable tests for service-oriented systems. Test modules in PASCANI are structured in three definition blocks: the system structure block, the test cases block, and the test control block. This block structure is a key factor for achieving reusability, composability, and traceability of test specifications. The objective of the system structure block is to define the services and component-based software structure to test, by importing not only system components, but also third-party services and other testing components, being this a mechanism that allows scalability in the definition of tests modules. The test cases and test control blocks allow to effectively reuse and compose testing components generated from test module specifications, through the invocation of imported services and other testing components. This reuse and composition is supported through the exposure and automatic binding of standard services for controlling traceability and test logging, which are generated at compile time. Our results show the
applicability of the framework by generating, deploying and executing SCA configurations of composable tests. Moreover, these results evidence the benefits that PASCANI offer, by saving considerable time and effort in the testing development process for human testers. Future work includes (i) the automatic generation of test modules to assemble, at runtime, test strategies for a specific target system; (ii) additional binding support beyond RMI and web service bindings; (iii) fully support for Linux platforms, the execution model of our current implementation is compatible only with Windows operating systems; and (iv) the integration of the framework to and IDE such as Eclipse to facilitate development, compilation, and test report visualization tasks.

ACKNOWLEDGEMENTS

This is a work in progress partially supported by Colciencias, the Colombian government agency for fundamental and applied research, under the I2T/DRISO Self-Adaptive Software Models and Design Patterns program. The authors would also like to thank the Research Office of Icesi University for supporting our participation in worldwide research networks.

REFERENCES


<table>
<thead>
<tr>
<th>Component</th>
<th>Test Module</th>
<th>PASCANI LOCs</th>
<th>Generated LOCs (Project) (XML)</th>
<th>Effort Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strassen</td>
<td>checkCorrectness</td>
<td>50</td>
<td>155</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>compareWithClassicStrassen</td>
<td>54</td>
<td>157</td>
<td>54</td>
</tr>
<tr>
<td>HybridMultiplication</td>
<td>checkCorrectness</td>
<td>45</td>
<td>143</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>checkExecutionTime</td>
<td>34</td>
<td>136</td>
<td>54</td>
</tr>
<tr>
<td>NMatrices</td>
<td>checkExecutionTime</td>
<td>30</td>
<td>127</td>
<td>54</td>
</tr>
<tr>
<td>MatrixChainMultiplication</td>
<td>compareMethods</td>
<td>35</td>
<td>479</td>
<td>181</td>
</tr>
<tr>
<td>CCNV</td>
<td>isServiceAlive</td>
<td>19</td>
<td>126</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>testCreditCardPartners</td>
<td>50</td>
<td>151</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>testCreditCardNumbersVerification</td>
<td>55</td>
<td>151</td>
<td>54</td>
</tr>
<tr>
<td>PAV</td>
<td>isServiceAlive</td>
<td>24</td>
<td>136</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>verifyClientAddress</td>
<td>51</td>
<td>134</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>verifyZipCodeCities</td>
<td>27</td>
<td>130</td>
<td>54</td>
</tr>
</tbody>
</table>

Table II: Relative effort gain when specifying tests with PASCANI