White-Box Testing of Service Compositions Via Event Interfaces

Chunyang Ye
University of Toronto
Toronto, Canada
chunyang.ye@utoronto.ca

Hans-Arno Jacobsen
University of Toronto
Toronto, Canada
jacobsen@eecg.toronto.edu

ABSTRACT

Service-oriented applications are usually composed of services from different organizations. To protect the business interests of service providers, the implementation details of services are usually invisible to service consumers. This makes it challenging to white-box test service-oriented applications because of the difficulty to determine accurately the test coverage of a service composition as a whole and the difficulty to design test cases effectively. To address this problem, we propose an approach to white-box test service compositions based on events exposed by services. By deriving event interfaces to explore test coverage information from service implementations, our approach allows service consumers to determine accurately test coverage during testing based on events exposed by services at runtime. We also develop an approach to design test cases effectively based on services' event interfaces. The experimental results show that our approach outperforms existing testing approaches for service compositions with 35% more test coverage rate, 19% more fault-detection rate and 80% fewer test cases needed.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Measurement, Reliability, Verification

Keywords

Web service composition, white-box testing, event interface

1. INTRODUCTION

The service-oriented architecture (SOA) paradigm is a widely adopted set of software engineering principles to help manage the complexity of software development for distributed enterprise applications [5, 11]. In this paradigm, service providers develop reusable software components, publish them as Web services, and register them in service registries. By composing selected services from registries, service consumers develop composite SOA applications across distributed, heterogeneous and autonomous organizations [15, 29].

To guarantee the quality of SOA applications, integration testing of service compositions is required before the applications are released. Testing is a challenging task, especially, when an SOA application integrates third-party services from different organizations. On the one hand, white-box testing of a service composition requires implementation details of every third-party service involved in the composition to be available [5, 17, 20]. However, for business reasons or privacy concerns, service implementation details must often remain hidden from service consumers. On the other hand, black-box testing [14, 22] requires no implementation details of services to become visible but suffers from the limitation that service consumers have little confidence on how well a service composition has been covered in testing [31].

To address this dilemma, Bartolini *et al.* proposed an approach which requires a service provider to report coverage information for its services to service consumers for testing purposes, such as the percentage of code paths covered, instead of revealing the services' implementation [1]. Based on the coverage information reported and the given coverage criteria, service consumers can estimate how well the involved services have been tested.

However, we observe that it is still difficult to apply this approach to a service composition involving more than one third-party service. There are two reasons for this (1) the inability to accurately determine test coverage as a whole and (2) the difficulty of effectively designing test cases.

First, although a service consumer can obtain the coverage percentage of every third-party service involved in a service composition under test, the service consumer is still unable to accurately determine how well the composition as a whole has been tested.

For example, as illustrated in Fig. 1, a manufacturer composes two third-party *item supplier* services to a *manufacturer* service. Each item supplier employs two different ways to produce items. Let us consider the following scenario: Suppose the first two test cases cover the two paths b_1, b_2, b_3, c_1, c_2, c_3 and b_1, b_4, b_5, c_1, c_4, c_5, respectively. According to the solution proposed in [1], the manufacturer then stops the testing and releases the service composition since all the involved services in Fig. 1 report 100% path coverage. However, the composition is not adequately tested because two other scenarios (i.e., paths b_1, b_2, b_3, c_1, c_4, c_5 and b_1, b_4, b_5, c_1, c_2, c_3) are not covered by this testing. Moreover, even with more test cases, the manufacturer still cannot determine whether all the scenarios are covered because all third-party services report 100% coverage. As a result,

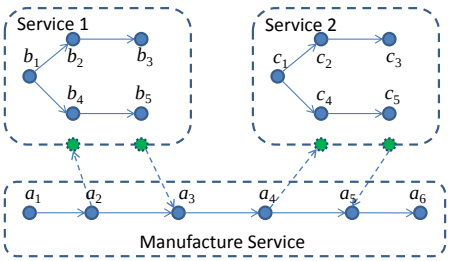


Figure 1: Coverage of a service composition.

faults in untested scenarios are left undetected (e.g., items produced via path b_1, b_2, b_3 may turn out to be inconsistent with items produced via path c_1, c_4, c_5).

Second, it is difficult for a service consumer to effectively design test cases to cover a service composition because of possible dependencies among services.

For example, suppose Service 2 in Fig. 1 executes the path c_1, c_4, c_5 if and only if Service 1 executes the path b_1, b_4, b_5 in the composition. The test cases designed to cover path c_1, c_4, c_5 in Service 2 will not work if Service 1 executes the path b_1, b_2, b_3 under these test cases. A dependency relationship such as this cannot be derived from the reported coverage percentages of Service 1 and 2 (as neither service has the required information). As a result, service consumers may need to try a large number of test cases to cover all possible scenarios, resulting in a significantly increased testing effort.

Therefore, reporting only the coverage percentage of each third-party service for testing is not enough. This motivated us to explore what other information services could reveal for testing while keeping their implementation details invisible from service consumers.

To address this concern, in this paper, we explore the potential of allowing services to expose events to support white-box testing of service compositions. In our approach, instead of reporting the coverage percentage, each service provider is required to provide service consumers with an event interface derived from the service implementation at design time. The event interface encapsulates and reveals selected service internal state changes as events at runtime.

For example, Service 1 in Fig. 1 may declare an event e_1 to reveal the status change of task b_2 (i.e., from "*non-committed*" to "*committed*") inside the service. Similarly, another event e_2 can be defined to reveal status changes of b_4. These events are correlated in the event interface to represent different executions of the service (e.g., e_1 and e_2 represents two different paths of Service 1). During testing, events exposed by third-party services are propagated to service consumers, who can then make use of them to determine test coverage of a service composition and effectively derive test cases.

There are two main challenges with this approach: (1) how to encapsulate and expose only the necessary events from a large number of events generated by services at runtime to hide the service implementation details; and (2) how to correlate events from different services to reason about the coverage of a service composition as a whole.

These challenges are addressed in this paper with a four-fold contribution: First, we propose a novel approach to white-box test Web service compositions involving more than one third-party service via events exposed from services. Next, we develop a model to derive event interfaces from service implementations. We prove that the test coverage derived based on event interfaces is equivalent to real coverage of service compositions under test. This allows service consumers to determine the test coverage of a service composition without revealing the implementation of each involved third-party service. Third, we propose algorithms to effectively derive test cases based on event interfaces to reduce the number of test cases needed. Finally, we perform a detailed experimental evaluation. The results show that our approach achieves a 35% increase in test coverage and detects 19% more faults than the approach proposed by Bartolini *et al.* [1], and requires 80% fewer test cases on average than the random testing approach [19].

The rest of this paper is organized as follows: Section 2 reviews related work on service testing. Section 3 introduces our approach and methodology. Section 4 evaluates our approach empirically and Section 5 discusses some limitations.

2. RELATED WORK

In this section, we review related work in the areas of service testing and service interfaces.

Service Testing. Service Testing has become an active area of research in the software engineering community and has attracted much attention in recent years [1, 5, 17, 20, 21]. Existing approaches can be classified into two main categories based on the roles involved in testing services: (1) from the perspective of service providers and (2) from the perspective of service consumers.

From the perspective of service providers, services need to be tested to conform to quality standards prior to release. Service providers usually have all the implementation details of their services (or partial details if third-party services are integrated to implement their services). Therefore, service providers can white-box test their services. For example, Li *et al.* [16, 30] proposed a framework to organize unit tests and generate test cases based on a search of BPEL flow graphs and constraint solving techniques. Mei *et al.* [20, 21] proposed a data flow approach to detect faults introduced by XML and XPath based on XPath rewriting. These approaches however are inadequate to white-box test a service composition involving third-party services because the implementations of third-party services are usually hidden.

Service consumers, on the other hand, need to know whether the selected third-party services work correctly when composed together to form new applications, even though each service has been tested individually by each service provider. Black-box testing approaches are often applied due to unavailable implementations of third-party services. Kaschner [14] proposed an automatic approach to design test cases for black-box testing of services based on their business protocols. Bartolini *et al.* [3] proposed a model-based approach to generate testbeds to replace services for testing from service consumers. Mei *et al.* [22] proposed an approach to help service consumers prioritize test case selection for regression testing based on the coverage of WSDL tags of the tested service. These approaches can help service consumers to detect faults in a service composition. The limitation is that it is unclear to service consumers how adequately a service composition as a whole has been tested.

To gain confidence about how well a service composition has been tested, service consumers need to whiten SOA testing for service compositions. Li *et al.* [17] suggested that service providers design test cases based on their BPEL processes and provide the test cases to service consumers. The limitation is that service providers cannot anticipate all possible composition scenarios. Bartolini *et al.* [1] proposed to instrument each service with an intermediate service which provides coverage feedback for each third-party service to service consumers during testing. However, the coverage percentage provided by such an approach can not be used to derive how adequately a whole service composition has been tested. Moreover, this approach does not address how to design test cases based on the feedback. We view this as a non-trivial step. Our approach addresses these two issues through the novel concept of event exposure from services

that we developed. By observing events and matching them to feasible observations constructed from event interfaces, service consumers can determine the test coverage of a service composition as a whole. Test cases can also be designed effectively based on event interfaces, as we demonstrate.

Testing Equivalence of Processes. Much research has also been devoted to conformance testing of service specifications and their implementations. For example, Nicola *et al.* [23] studied the equivalence relationship between processes based on a set of tests. Bentakouk *et al.* [2] proposed to test the conformance between service orchestration specifications and their implementations with symbolic execution techniques. Tretmans [26] defined a test equivalence relationship between asynchronous input/output automata and the underlying synchronous labeled transition systems.

Our work also defines an equivalence relationship between the event interface and the service implementation. The difference is that existing work does conformance testing of service specification and implementation, whereas our approach determines the test coverage for white-box testing. Even though two processes are equivalent in terms of conformance testing, the test cases designed based on one process may not cover the same paths in the other process.

Service Interface. Often, the implementation details of a service are invisible to service consumers except for access to restricted service interfaces. Many researchers have studied how to enrich service interfaces to facilitate a service composition. Beyer *et al.* [4] proposed to specify constraints in Web service interfaces to define correctness requirements of a service. Alfaro and Henzinger [8] proposed to describe interfaces as automata to capture temporal aspects of constraints. Emmi *et al.* [9] proposed a modular verification approach based on assume-guarantee rules to check the compatibility of interface automata. Ye *et al.* [29] proposed an atomicity-equivalent public view to check the atomicity property of a service composition. Some industrial standards like the SCA Event Interface [25] and the WS-Eventing protocol [27] were proposed to expose and propagate events among services. However, all aforementioned approaches do not address how to use service interfaces to white-box test a service composition. Our work contributes a new kind of service interface, namely an event interface, to test services and thus complements existing approaches.

3. METHODOLOGY

3.1 Overview

As discussed in Section 1, existing approaches to white-box testing service compositions suffer from two limitations: inability to accurately determine test coverage as a whole and difficulty to effectively design test cases. In this section, we illustrate our approach to address these issues based on the exposure of events from services.

An event is defined as a state change [6, 18]. A state of a service is defined as a snapshot of its execution at runtime. The execution of a service can be seen as a series of transitions among its states. The transition from one state to another is defined as a *state change*. For example, an online shopping service transitions from the state "*the customer has not been verified*" to "*the customer has been verified*". Usually, these states are invisible from outside the service, and thus referred to as *internal states*. We define an event to reveal a state change from within a service.

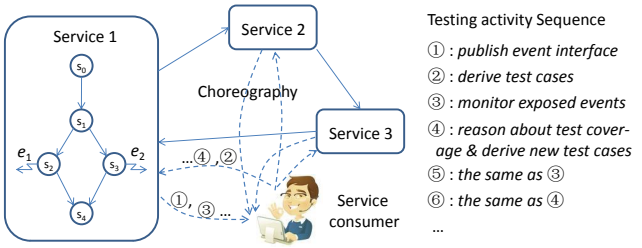


Figure 2: Methodology overview

In this paper, we explore the use of event exposure from services to support white-box testing of service compositions. The basic idea is to abstract coverage-related internal state changes as events and expose them to service consumers. For example, as illustrated in Fig. 2, suppose Service 1 transitions from state s_1 to state s_2, then we define and expose the event e_1 to represent that the path from s_1 to s_2 has been covered. Similarly, another event e_2 is defined to represent the coverage of the path from s_1 to s_3. By making use of coverage-related events, service consumers can accurately determine the test coverage of a service composition as a whole. The conditions under which these events occur can also be explored to help derive test cases more effectively to cover a service composition under test.

Note that given various coverage criteria, we can define different sets of events to represent the coverage scenario. For example, for a data flow coverage criterion to cover all define-use relations [31], we can define a pair of events (e_{def}, e_{use}) to track every define-use pair in the service. To ease the presentation and without lose of generality, in the rest of this paper, we use path coverage [31] to illustrate our approach. Other coverage criteria can be handled in a similar way.

Fig. 2 summarizes our methodology. Each third-party service provider defines coverage-related events in its service, abstracts them and their relationships into an event interface, and publishes the event interface to the service consumer. By monitoring and correlating the exposed events from third-party services during testing, the service consumer can determine how well the service composition has been tested. Additionally, the service consumer can use event interfaces to derive test cases to cover untested paths.

3.2 Coverage-equivalent Event Interface

To make use of event exposure from services to support white-box testing of service compositions, service providers need to encapsulate events related to test coverage, derive their relationships, and declare them in event interfaces. Before illustrating how to do so, let us first introduce some basic concepts. Similar to many existing work [4, 10, 11, 29], we model a service as a finite state machine in this paper. Each *state* is defined by a set of variables and their values. In Section 5, we discuss how to derive the state machine of a service from its implementation (e.g., BPEL).

Definition 1 (State): A state s is defined as a finite set $\{(x_1, t_1, v_1), \cdots, (x_n, t_n, v_n)\}(n > 0)$, where x_i is a variable, t_i and v_i are its type and value, respectively[1].

Definition 2 (Service): A service P is a 6 tuple $(S, s_0, G, C \cup I, T, F)$, where S is a set of states, $s_0 \in S$ is the initial state, $F \subseteq S$ is the set of final states, C is the set

[1]Note that the value of a variable can be a concrete value or a constraint to define a set of values (e.g., $v_i \equiv x_i > 0$).

of communicating actions (e.g., sending or receiving a message), I is the set of internal actions invisible to service consumers, G is the set of guarded Boolean expressions, and $T \equiv S \times G \times (C \cup I) \times S$ represents the set of transitions.

Given a state s in service P, P can transition from s to s', denoted as $s \xrightarrow{t} s'$, if and only if $\exists t \equiv (s, g, a, s') \in T \wedge s \vdash g$. An execution of P (also denoted as an *instance* of P) is a sequence of $s_0 \xrightarrow{t_{i1}} s_{i1} \xrightarrow{t_{i2}} \cdots \xrightarrow{t_{ik}} s_{ik}$, where s_{ik} is its current state. The execution of a service is the transitioning of the service from one state to another. Informally speaking, these state changes represent that something happened and are defined as events.

Definition 3 (Event): Let $s \equiv \{(x_1, t_1, v_1), \cdots, (x_n, t_n, v_n)\}$ and $s' \equiv \{(x'_1, t'_1, v'_1), \cdots, (x'_n, t'_n, v'_n)\}$ be two states of service P. A state change from s to s' is defined as an event $e_{s \to s'} \equiv \{(x_{i1}, t_{i1}, v_{i1}), \cdots, (x_{ik}, t_{ik}, v_{ik})\} \subseteq s \cup s'$.

Note that an event is different from a transition, in the sense that the former defines that something of interest happens (i.e., a state change related to a set of variables $x_{i1}, \cdots, x_{ik}$ of interest), whereas the latter defines how something happens (i.e., how a state change comes about). Since our purpose is to determine the coverage of a service composition during testing without revealing the implementation details, we only need to know the coverage changes during testing. Therefore, we define two kinds of events in our approach: *coverage-related events* and *auxiliary events*. The former are defined and raised to reflect the coverage changes for testing, whereas the latter are defined to correlate events from different services involved in a service composition.

To define coverage-related events, we introduce an extra variable $x_{coverage}$ into a service to collect the coverage information of a service for testing. For the path coverage criteria [31], this variable is assigned a different value (e.g., a unique branch ID) whenever a service enters a branch. For example, in Fig. 2, if Service 1 transitions from s_1 to s_2 (or s_3), $x_{coverage}$ can be assigned "*Branch 1*" (or "*Branch 2*"). Formally, suppose service P transitions from s to s', where $(s, g_1, a, s') \in T$, the value of $x_{coverage}$ changes if and only if $\exists (s, g_2, b, s'') \in T$: $(s, g_2, b, s'') \neq (s, g_1, a, s')$. Whenever the value of $x_{coverage}$ changes, we raise an event to represent such a change. For example, in Fig. 2, two events e_1 and e_2 are defined to indicate that Service 1 enters two different branches, respectively.

Besides coverage-related events, we also need to define some auxiliary events to correlate events from different services. An auxiliary event occurs when a service sends or receives a message: that is, if service P transitions from s to s', where $(s, g, a, s') \in T \wedge a \in C$, then an auxiliary event $e_{s \to s'}$ is defined. If both an auxiliary event and a coverage-related event are defined for a transition, then only the auxiliary event is kept. We also define a start event for each service to indicate that the service has started to execute.

In order to represent the actual coverage of paths inside a service, what is still needed is a way to determine which events are on the same path and which are not. For example, as illustrated in Fig. 3(a), suppose service P transitions from s_0 to s_4 via s_2, event e_0 will be raised first, followed by events e_1, e_2. Therefore, we can correlate e_0, e_1 and e_2 in a sequence $e_0 e_1 e_2 \cdots$ to represent the path. On the other hand, transitions $t_1 \equiv (s_1, g_1, a_1, s_2)$ and $t_8 \equiv (s_0, g_8, a_8, s_7)$ never belong to the same path in any execution of P. Therefore, e_2 and e_6 should not be correlated. The following definition summarizes the causality relationships among events:

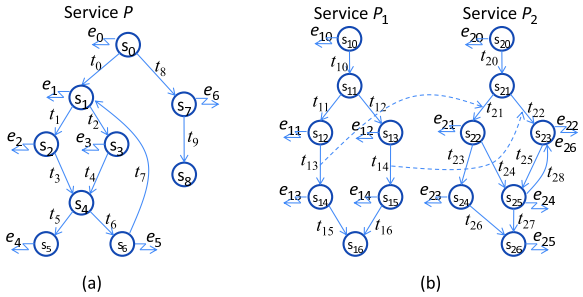


Figure 3: (a) Event exposure. (b) Service composition.

Given two events $e_{s_1 \to s_2}$ and $e_{s_3 \to s_4}$, $e_{s_1 \to s_2}$ is said to cause $e_{s_3 \to s_4}$, denoted as $C(e_{s_1 \to s_2}, e_{s_3 \to s_4})$, if and only if $\exists s_{i1} \xrightarrow{t_1} s_{i2} \xrightarrow{t_2} \cdots \xrightarrow{t_{ik}} s_{ik} \wedge s_{i1} \equiv s_2 \wedge s_{ik} \equiv s_3$. If no event is raised during $s_{i1} \xrightarrow{t_1} s_{i2} \xrightarrow{t_2} \cdots \xrightarrow{t_{ik}} s_{ik}$, $e_{s_1 \to s_2}$ is called the *direct cause* of $e_{s_3 \to s_4}$, denoted as $DC\ (e_{s_1 \to s_2}, e_{s_3 \to s_4})$. In the above example in Fig. 3(a), e_0, e_1 and e_2 cause e_4 in the path from s_0 to s_5 via s_1, s_2, s_4. Event e_2 is the direct cause of e_4 whereas e_0 and e_1 are not.

Based on the above discussion, we introduce the concept of *event interface* to abstract both the exposed events and their causality relationships inside a service.

Definition 4 (Event Interface): An event interface EI is a tuple (E, R), where E is the set of exposed events, and $R \equiv E \times E$ is the set of causality relationships between events, that is, $\forall (e_i, e_j) \in R$: $DC\ (e_i, e_j)$.

Since our purpose is to use event interfaces to determine test coverage inside services, service providers need to offer event interfaces for their services to service consumers before testing. The following algorithm illustrates how to derive an event interface from a service. As proven in Section 3.3, the coverage derived based on event interfaces is equivalent to the actual coverage of services in testing. Therefore, event interfaces are called *coverage-equivalent* event interfaces.

Algorithm 1 has two parts. Part 1 (Lines 2 to 18) traverses the service and generates the two types of events (Lines 6, 11); Part 2 (Lines 19 to 29) traverses the service in the opposite direction to determine the direct causes for each event. For example, as marked in Fig. 3(a), seven events $\{e_0, e_1, e_2, e_3, e_4, e_5, e_6\}$ are defined and exposed from Service P. For each event, the algorithm traces back from the state the event is raised to determine all potential direct causes. For instance, for event e_4, the algorithm traces back from state s_4 to s_2 and s_3, and gets its two possible direct causes e_2 and e_3. The causality relationship for other events can be calculated in a similar way. Therefore, the event interface for service P is $EI \equiv (E, R)$, where $E \equiv \{e_0, e_1, e_2, e_3, e_4, e_5, e_6\}$, $R \equiv \{(e_0, e_1), (e_0, e_6), (e_1, e_2), (e_1, e_3), (e_2, e_4), (e_2, e_5), (e_3, e_4), (e_3, e_5), (e_5, e_1)\}$. Suppose a service has k transitions, and exposes m events. Part 1 (Lines 2 to 18) traverses at most k steps; Part 2 (Lines 19 to 29) traverses at most $m \times k$ steps. Since $m \leq k$, the complexity of Algorithm 1 in the worst case is $O(k^2)$.

3.3 Coverage Reasoning

Based on the event interfaces provided by service providers, service consumers can monitor the exposed events at runtime to determine test coverage. As mentioned in Section 1, the execution of a path in one service may depend on some

Algorithm 1 Derive a coverage-equivalent event interface.

Input:
A service $P \equiv (S, s_0, G, C \cup I, T, F)$;
Output:
A coverage-equivalent event interface $EI \equiv (E, R)$;

$q_{search} \leftarrow \{s_0\}$, $E \leftarrow e_0$, $generated(s_0) \leftarrow e_0$;
while $\exists cs \in q_{search}$ **do**
  $q_{search} \leftarrow q_{search} - \{cs\}$;
  for $\forall t \equiv (cs, g, a, s) \in T$ **do**
    if $a \in C$ **then**
      define an auxiliary event e_a;
      $E \leftarrow E \cup \{e_a\}, cause(e_a) \leftarrow cs$;
      $generated(s) \leftarrow generated(s) \cup \{e_a\}$;
    else
      if $\exists t' \equiv (cs, g', a', s') \in T : t \neq t'$ **then**
        define a coverage-related event e_c;
        $E \leftarrow E \cup \{e_c\}$, $cause(e_c) \leftarrow cs$;
        $generated(s) \leftarrow generated(s) \cup \{e_c\}$;
      else
        $shared(s) \leftarrow shared(s) \cup \{cs\}$;
    if $visited_{r1}(s) = false$ **then**
      $q_{search} \leftarrow q_{search} \cup s$;
      $visited_{r1}(s) \leftarrow true$;
for $\forall e \in E$ **do**
  $s \leftarrow cause(e)$;
  if $visited_{r2}(s) = false$ **then**
    $visited_{r2}(s) \leftarrow true, q_{search} \leftarrow \{s\}$;
    while $\exists cs \in q_{search}$ **do**
      $q_{search} \leftarrow q_{search} - \{cs\}$;
      $cause\_set(s) \leftarrow cause\_set(s) \cup generated(s)$;
      for $\forall ns \in shared(cs) : visited_{r2}(ns) = false$ **do**
        $q_{search} \leftarrow q_{search} \cup \{ns\}$;
  for $\forall e' \in cause\_set(s)$ **do**
    $R \leftarrow R \cup \{(e', e)\}$

particular paths in another service. Suppose n services are involved in a service composition (denoted as P), and each one has m_i paths, then P may have a total of $\prod_{i=1}^{n} m_i$ possible combinations of execution paths. To accurately determine test coverage of P as a whole, service consumers need to know which combinations of execution paths are feasible. Definition 5 specifies feasible paths in a service composition.

Definition 5 (Service Composition): Given n services $P_i \equiv (S_i, s_{i,0}, G_i, C_i \cup I_i, T_i, F_i)(i = 1..n)$, their composition is denoted as a service $P \equiv \oplus(P_1, P_2, \cdots, P_n)$. A state of P can be represented as $((s_{1,j_1}, w_1), \cdots, (s_{n,j_n}, w_n))$, where $s_{i,j_i} \in S_i$, and w_i is a sequence of executed actions in $C_i \cup I_i$ representing a path. The transition $((s_{1,j_1}, w_1), \cdots, (s_{n,j_n}, w_n)) \rightarrow ((s'_{1,j_1}, w'_1), \cdots, (s'_{n,j_n}, w'_n))$ is allowed if and only if any of the following conditions are satisfied:

1. $\exists t_i \equiv (s_{i,j_i}, g_i, a_i, s'_{i,j_i}) : s_{i,j_i} \xrightarrow{t_i} s'_{i,j_i} \wedge a_i \in I_i \wedge w'_i = w_i a_i \wedge (\forall l \neq i : s'_{l,j_l} = s_{l,j_l} \wedge w'_l = w_l)$.
2. $\exists t_i \equiv (s_{i,j_i}, g_i, a_i, s'_{i,j_i}) : (s_{i,j_i} \xrightarrow{t_i} s'_{i,j_i} \wedge a_i \in C_i \wedge w'_i = w_i a_i \wedge \exists t_k \equiv (s_{k,j_k}, g_k, a_k, s'_{k,j_k}) : (s_{k,j_k} \xrightarrow{t_k} s'_{k,j_k} \wedge a_k \in C_k \wedge w'_k = w_k a_k \wedge (a_i$ sending a message and a_k receiving the message) $\wedge (\forall l \neq i, k : s'_{l,j_l} = s_{l,j_l} \wedge w'_l = w_l)))$.

Intuitively, Definition 5 specifies how a service composition transitions from one state to another. In particular, Condition 1 represents a transition by executing an internal action (that is, from I_i) of an involved service P_i; Condition 2 represents a transition that two involved services P_i and P_k communicate with each other via a_i and a_k[2]. A state $((s_{1,j_1}, w_1), \cdots, (s_{n,j_n}, w_n))$ is *feasible* if and only if there exists a sequence of transitions $((s_{1,0}, \{\}), \cdots, (s_{n,0}, \{\})) \rightarrow \cdots \rightarrow ((s_{1,j_1}, w_1), \cdots, (s_{n,j_n}, w_n))$.

Let us take the service composition in Fig. 3(b) as an example. Actions in transitions t_{13} and t_{14} of service P_1 send two messages to P_2 (represented as the dashed curve). These two messages are received by t_{21} and t_{22} of P_2, respectively. The initial state of the service composition is $((s_{10}, \{\}), (s_{20}, \{\}))$. If P_1 transitions to s_{14}, P_2 will transition to s_{22}. Therefore, the state $((s_{14}, a_{10}a_{11}a_{13}), (s_{22}, a_{20}a_{21}))$ is feasible, where a_i represents the action in transition t_i. Similarly, $((s_{15}, a_{10}a_{12}a_{14}), (s_{23}, a_{20}a_{22}))$ is also feasible whereas $((s_{14}, a_{10}a_{11}a_{13}), (s_{23}, a_{20}a_{22}))$ is not, because when P_1 transitions to s_{14}, P_2 cannot transition to s_{23}.

A feasible state of a service composition represents a feasible execution path of the service composition. To accurately determine test coverage of a service composition as a whole, service consumers need to correlate events in event interfaces from different services and organize them in a way to enumerate every feasible execution path of the service composition. On the other hand, every feasible execution path of the service composition should have only one such combination of events. Definition 6 specifies the pattern to organize and correlate events to represent a feasible execution path of a service composition. To ease the presentation, we introduce the following notation: Given a sequence of events $h \equiv e_0 e_1 \cdots e_n$, $tail(h) \equiv e_n$ and $he \equiv e_0 e_1 \cdots e_n e$; given an event e, predicates $au_s(e), au_r(e), cv(e)$ denote that e is an auxiliary event representing the sending of a message, an auxiliary event representing the receiving of a message, and a coverage-related event, respectively; $com(e_i, e_j) = true$ if and only if e_i is the event representing that a message is sent by a service and e_j is the event representing that this message is received by another service.

Definition 6 (Observation): Let P be a service composition of n services $P_i(i = 1..n)$, and $EI_i \equiv (E_i, R_i)$ be their coverage-equivalent event interfaces. An observation of events from this service composition can be represented as $(h_1, h_2, \cdots, h_n)$, where h_i is a sequence of events observed from service P_i. The observation $(h_1, h_2, \cdots, h_n)$ can be followed by $(h'_1, h'_2, \cdots, h'_n)$, denoted as $(h_1, h_2, \cdots, h_n) \Rightarrow (h'_1, h'_2, \cdots, h'_n)$, if and only if $\exists e_j \in E_i : h'_i = h_i e_j \wedge (tail(h_i), e_j) \in R_i \wedge (\forall l \neq i : h'_l = h_l) \wedge (au_s(e_j) \vee cv(e_j) \vee (\exists k \neq i : com(tail(h_k), e_j))$.

The intuitive meaning of Definition 6 is that a service consumer can observe a new event from a service if and only if its direct cause is observed as the latest event from the service. Moreover, if the event indicates that a message is received, then the event indicating that the same message is sent by a service should be the latest observed event from the service[3]. An observation $(h_1, h_2, \cdots, h_n)$ is *feasible* if and only if there exists a sequence $(e_{1,0}, e_{2,0}, \cdots, e_{n,0}) \Rightarrow \cdots \Rightarrow (h_1, h_2, \cdots, h_n)$, where $e_{i,0}$ is the start event of P_i. For example, as illustrated in Fig. 3(b), $(e_{10}e_{11}e_{13}, e_{20}e_{21})$ is a feasible observation whereas $(e_{10}e_{11}e_{13}, e_{20}e_{22})$ is not. Based on the above concepts, we have the following theorem:

[2]We assume synchronous communication between services. We discuss how to handle asynchronous communication in Section 5.

[3]Events may lose order in a distributed setting. Service consumers can solve this issue by caching events into queues before matching them to the observed pattern of Definition 6.

THEOREM 1. *Let P be a service composition of n services $P_i(i = 1..n)$, and $EI_i \equiv (E_i, R_i)$ be their coverage-equivalent event interfaces. For every feasible execution $((s_{1,j_1}, w_1), \cdots, (s_{n,j_n}, w_n))$ of P, there exists a feasible observation $(h_1, h_2, \cdots, h_n)$ of events from P, and vice versa, where h_i is the sequence of events generated by P_i in the execution.*

The intuitive meaning of Theorem 1 is that every feasible observation of events corresponds to a feasible execution path of a service composition, and vice versa. The proof of Theorem 1 is to construct a one-to-one mapping between a feasible execution path of a service composition and a feasible observation of events. The proof can be found in the appendix.

With this theorem, service consumers only need to construct all the feasible observations based on the event interfaces from service providers, and determine the test coverage by counting how many of them have been observed during testing. Note that when there are loops in a service composition, the number of feasible execution paths may be infinite. In practice, some constraints are usually added to the path coverage criterion to terminate searching (e.g., the length of each path is less than a given K [31], and this information can be added into events). On the other hand, verifying a path is executable or not is generally undecidable. As a result, white-box testing techniques usually count how many of the potentially executable paths (whose executable conditions may be satisfied) have been covered [31]. Consequently, the following algorithm constructs all the corresponding potentially feasible observations.

Algorithm 2 Construct all feasible observations.

Input:
n event interfaces $EI_i \equiv (E_i, R_i)$ of n services involved in a service composition;
Output:
A set of potential feasible observations $O \equiv \{(h_{1,1}, h_{2,1}, \cdots, h_{n,1}), \cdots, (h_{1,m}, h_{2,m}, \cdots, h_{n,m})\}$;

```
1: q_search ← {(e_{1,0}, e_{2,0}, ··· , e_{n,0})}, O ← {};
2: while ∃co ∈ q_search do
3:   q_search ← q_search − co(≡ (h_1, h_2, ··· , h_n));
4:   visited(co) ← true;
5:   if satisfied(co) then
6:     O ← O ∪ {co};
7:   else
8:     for ∀i(i = 1..n) do
9:       for ∀e ∈ E_i : tail(h_i, e) ∈ R_i do
10:        if cv(e) ∨ aus(e) ∨ (∃j : com(tail(h_j), e)) then
11:          co' = (h_1, ··· , h_i e, ··· , h_n);
12:          if visited(co') = false then
13:            q_search ← q_search ∪ {co'};
```

Algorithm 2 constructs all potentially feasible observations from scratch. In the beginning, only the start events of each involved service are put into an initial observation. Then, the algorithm constructs all the potentially feasible observations following the initial observation by adding an event to the observation that may satisfy the condition in Definition 6. If a new potentially feasible follow-up observation satisfies the requirement (Line 5, e.g., the length of the path is larger than a given K), the observation is put into the output set O. This procedure is executed until no more potentially feasible observations are found (Line 2). Suppose a service composition has m feasible execution paths, Algorithm 2 executes at most $m \times K$ steps. Therefore, the complexity of this algorithm is $O(m \times K)$.

3.4 Test Case Generation

As mentioned in Section 3.3, service consumers can construct all the potentially feasible observations based on the coverage-equivalent event interfaces from service providers. During testing, service consumers generate test cases to test service compositions and subscribe to the exposed events from involved services. By counting the number of potentially feasible observations that have been matched from exposed events, a service consumer can determine how well a service composition has been tested.

One important issue remained is how to design test cases to cover the potentially feasible observations. Service consumers can apply existing approaches (e.g., random testing [19]) to generate test cases. However, as mentioned in Section 1, the execution of a path in one service may depend on the execution state of some particular paths in another service. A dependency such as this increases the difficulty to generate test cases effectively to cover a service composition adequately. In this section, we illustrate that the dependency information can be explored and attached to event interfaces. The purpose is to allow service consumers to use the additionally exposed information to generate test cases effectively to cover potentially feasible observations. Note that the test oracle issue is out of the scope of this paper. The approach introduced in this section can be seen as a complementary approach to existing test case generation approaches for service testing.

One natural solution to the aforementioned issue is to add more information related to exposed events to provide more insights about the internal execution of a service. In particular, we can analyze the service to determine the conditions under which an event can be raised in a given potentially feasible observation. For example, as illustrated in Fig. 4, event e_{11} is raised if and only if service P_1 transitions from s_{10} to s_{12}, that is, $s_{10} \xrightarrow{t_{10}} s_{11} \xrightarrow{t_{11}} s_{12}$. Since the guarded condition of t_{10} is always *true*, P_1 can transition from s_{10} to s_{11}. According to the action in t_{10}, state s_{11} should satisfy the following condition: $y = x + 10$. If $s_{11} \xrightarrow{t_{11}} s_{12}$, then $s_{11} \vdash g_{11}$ should be satisfied, that is, the condition $(y = x + 10) \wedge (y < 20)$ should be satisfied at state s_{11}. Therefore, under this condition, e_{11} is raised to follow after e_{10}. To ease the presentation, we call this condition the *causality condition* between events e_{10} and e_{11}, denoted as $CC(e_{10}, e_{11})$.

Note that the causality conditions between events can be derived during the construction of event interfaces in Algorithm 1. Formally, suppose $s \xrightarrow{t} s'$ and $SC(s)$ represent the constraints s satisfies, then $SC(s') \equiv SC(s) \wedge g \wedge a$, where g and a are the guarded condition and action in t, respectively[4]. We can iteratively apply this rule during the traversal of a service in Algorithm 1 to calculate the causality conditions between every exposed event and its direct cause. Suppose event $e_{s_1 \to s_2}$ is the direct cause of $e_{s_k \to s_{k+1}}$, that is, $\exists s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots \xrightarrow{t_k} s_{k+1}$ and no event is raised during $s_2 \xrightarrow{t_2} s_3 \xrightarrow{t_3} \cdots \xrightarrow{t_{k-1}} s_k$. $CC(e_{s_1 \to s_2}, e_{s_k \to s_{k+1}})$ is equiv-

[4]We can always rename the variables to make sure each variable is assigned the value only once. Therefore, we can calculate the constraint of $SC(s')$ in this way.

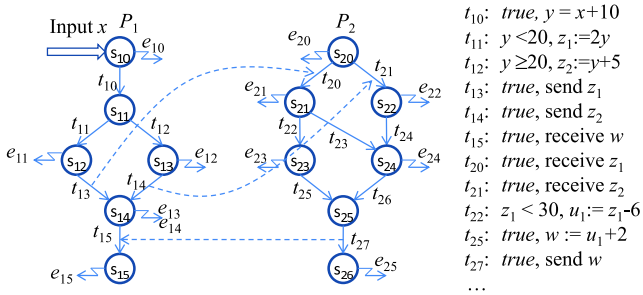


Figure 4: Causality conditions for events.

alent to $SC(s_1) \wedge (\bigwedge_{j=1}^{k}(g_j \wedge a_j))$ where g_j and a_j are the guarded condition and action in transition t_j, respectively.

Let $EI \equiv (E, R)$ be an event interface for service P, EI can be extended to include the local causality conditions between events, that is, $EI' \equiv (E, R')$, where $\forall(e_1, e_2) \in R : (e_1, e_2, CC(e_1, e_2)) \in R'$, and vice versa. By attaching the local causality conditions into the event interfaces as well, service consumers can integrate them into global constraints for the service composition and apply constraint solving techniques to generate the test cases for each given feasible observation. An alternative approach is that each involved service keeps its own local causality conditions invisible to service consumers and collaborates with its partner services to generate the test cases based on their own causality conditions. In this paper, we illustrate the former and leave the latter as future work.

Similar to the calculation of causality conditions between events, the global constraints for a potentially feasible observation can be calculated iteratively. Let $EI'_i (i = 1..n)$ be the extended event interface of service P_i, and $o \equiv (h_1, \cdots, h_n)$ be a potentially feasible observation, the global constraint that must be satisfied for o is denoted as $GC(o)$. Suppose $o \Rightarrow o'$, where $o' \equiv (h_1, \cdots, h_i e, \cdots, h_n)$, then $GC(o')$ $\equiv GC(o) \wedge CC(tail(h_i), e)$. For example, given a potentially feasible observation $o_1 \equiv (e_{10}e_{11}e_{13}e_{15}, e_{20}e_{21}e_{23}e_{25})$ in Fig. 4 and the causality conditions among the events in o_1, $GC(o_1) \equiv (y = x+10) \wedge (y < 20) \wedge (z_1 = 2y) \wedge (z_1 < 30) \wedge (u_1 = z_1 - 6) \wedge (w = u_1 + 2)$. When $GC(o_1)$ is satisfied, o_1 can be observed by matching the exposed events from the service composition. By applying constraint solving techniques to $GC(o_1)$, service consumers can obtain a solution $\{x = 2, y = 12, z_1 = 24, u_1 = 18, w = 20\}$. This solution indicates a test case (that is, $\{x = 2\}$) to cover the feasible observation o_1.

In practice, a test case may involve many interactions between the service consumer and the service composition being tested. For example, a customer needs to input the query condition, receive the query result, input the confirmation etc. We can model the test case as a service involved in the service composition as well. The generation of a test case is the solution to the local variables inside the test case service satisfying the global constraints.

4. EVALUATION

As illustrated in Section 3, our approach makes use of events exposed by services and event interfaces to determine test coverage of a service composition as a whole and to derive test cases for the composition. This section evaluates the approach quantitatively by comparing it to existing work in terms of coverage rate, effectiveness in fault-detection and test case generation. We also evaluate the running time complexity of our algorithms and overhead for event exposure.

4.1 Experimental Setup

We use three open-source service compositions to evaluate our work: A supply-chain application [28] (denoted as SC), a loan approval application [13] (denoted as LA) and a book ordering application [24] (denoted as BO). Each application is characterized in Table 7 by listing the number of services, states, transitions, and events exposed for our approach. These applications are also used for service testing by others [2, 14, 20, 30].

Table 1: Applications and descriptive statistics

Services		#States	#Trans.	#Paths	#Events
SC	s_1	15	19	6	12
	s_2	12	15	5	7
	Comp.	25	30	18	19
LA	s_1	5	7	3	8
	s_2	8	9	3	8
	s_3	6	8	4	10
	Comp.	24	29	10	26
BO	s_1	14	17	5	11
	s_2	10	13	5	9
	Comp.	24	29	15	20

In the first experiment, we evaluate the coverage percentage in testing and the effectiveness in fault detection of our approach. We use the approach proposed by Bartolini *et al.* as a baseline [1]. In the baseline approach, we leverage the testing based on the coverage percentage of each involved third-party service. We compare the coverage percentage of a service composition as a whole in testing using our approach (denoted as OA) and the baseline approach (denoted as EA). To evaluate the effectiveness in fault detection, we measure and compare the fault-detection rate [12] of both approaches.

To evaluate the fault-detection rate, faulty versions of service compositions are needed. However, to the best of our knowledge, few faulty versions are reported by developers. Therefore, we generate different faulty versions of service compositions by seeding one fault into the three original service compositions following the guidelines in [12]. To be fair, we seed two types of faults: Faults of Type 1 are internal to a service (e.g., missing functionality), and usually can be detected by unit testing of the service; faults of Type 2 represent integration faults that are caused by inconsistency among services (e.g., inconsistent items produced by Services 1 and 2 in Fig. 1). Faults of Type 2 are usually specific to some particular paths across different services in a service composition. In total, we create 30 faulty versions (Type 1: SC(6), LA(4), BO(5) and Type 2: SC(6), LA(4), BO(5)). Detailed description of services and seeded faults can be found in the appendix.

We then generate test suites for our approach and the baseline approach. We randomly select a test case from a test pool and execute a target version of a service composition over the test case. If the test case improves the coverage percentage reported by OA or EA[5], then it is added to the

[5]Note that the coverage percentage reported by our approach is equivalent to the ratio of the number of feasible observations observed against the total number of potentially feasible observations; whereas the value reported by

test suite for the corresponding approach. The test case selection procedure terminates if 100% coverage is achieved with the maximum length of a path set to 100, or if after a maximum number (500) of trials, the coverage is not improved. This procedure is repeated 2,000 times for each version. The fault-detection rate is calculated as the ratio of the number of test suites that can detect the fault in the version against the total number of test suites selected. The real coverage percentage calculated in testing is equivalent to the ratio of the number of executed paths against the total number of potential paths in the service composition.

In the second experiment, we evaluate the effectiveness of test case generation for service compositions. We use the random testing approach [19] as a baseline, that is, test cases are randomly generated to test a service composition. To evaluate the effectiveness of test case generation, we measure the number of test cases needed to cover each service composition with differently given coverage percentages (cover the paths of the service composition as a whole). The test case generation procedure terminates if 100% coverage is achieved, or if after a maximum number (200) of trials the coverage is not improved. Whenever the coverage percentage is updated, the total number of test cases needed to reach the coverage percentage is recorded. The test case generation procedure is repeated 100 times for each service composition. The number of test cases needed in both approaches is compared.

Finally, we evaluate the complexity of our algorithms. We randomly generate a set of services with the number of states varying from 1,000 to 10,000, and apply our algorithms to derive event interfaces. The overhead for exposing events at runtime for each service is also recorded. We also construct potentially feasible observations for randomly generated service compositions with the number of states varying from 1,000 to 10,000. The experiment is repeated 1,000 times and the average time needed for both algorithms and runtime overhead for event exposure is recorded.

4.2 Experiment Data Analysis

In this section, we analyze and report the experimental results. In the first experiment, the minimum, mean, and maximum coverage percentage during the testing of both approaches are shown in Fig. 5(a). In each case, our approach has better coverage percentage than the existing approach. In particular, our approach has 15%, 40%, and 40% higher coverage percentage than the existing approach for the application LA, 14%, 35%, and 40% for SC, and 15%, 17%, 23% for BO, respectively.

The fault-detection rates for each category of faults and the aggregated results are shown in Fig. 5(b). The results show that our approach has a much higher fault-detection rate than the existing approach, especially for faults of Type 2. In particular, with respect to fault-detection rate for faults of Type 1, Type 2 and overall, our approach achieves 0.16, 0.25, and 0.20 more than the existing approach for the application LA, and 0.05, 0.41, and 0.23 more for SC, and 0.09, 0.17, and 0.13 more for BO, respectively.

Since the drop in coverage percentage and effectiveness of EA may be due to fewer test cases in its test suites, we randomly added some extra test cases to the test suites in EA to make sure that the number of test cases is equivalent to that in OA. We repeated the experiment 2,000 times, and the results are shown in Fig. 5(c) and Fig. 5(d). Now, the average coverage percentage of the existing approach is improved, but our approach still achieves at lease 10%, 16%, and 20% more coverage percentage than the existing approach for the minimum, mean, and maximum cases, respectively.

For the overall fault-detection rate, our approach still achieves 0.02, 0.13, and 0.12 higher as compared to the existing approach for the LA, SC, and BO application scenarios, respectively. However, the fault-detection rate for faults of Type 1 achieved by the existing approach is a little higher (0.02 and 0.05) than that of our approach for LA and SC, although the coverage percentage of the existing approach is lower than that of ours. This may be because faults of Type 1 are local to certain individual services in a service composition. By randomly adding extra test cases to the existing approach, all paths of a service can be covered with similar probability. However, in our approach, the possibility of covering a path in a service depends on other services in a service composition. As a result, certain paths in a service may be covered with more probability whereas the others are covered with less probability. Therefore, faults of Type 1 are more likely to be discovered by a test suite that covers all the paths of a service evenly in the existing approach. The fault-detection rate for faults of Type 2 in our approach is still higher than the existing approach (that is, 0.06, 0.29, and 0.16 for the LA, SC and BO applications, respectively). This is because faults of Type 2 are across different services, and our approach achieves a higher coverage percentage than the existing approach. As a result, faults of Type 2 are more likely to be discovered using our approach. This result implies that our approach is more useful to detect integration faults in a service composition.

For the second experiment, Fig. 5(e) illustrates the number of test cases needed using both our approach (denoted as OA) and the random generation approach (denoted as EA-RD) to achieve the given coverage percentage in all the three applications. The figure shows that the number of test cases needed using the random testing approach increases dramatically when the coverage percentage increases, whereas the number of test cases needed in our approach is much smaller (average 80% less) than that of the existing approach and nearly linear to the coverage percentage.

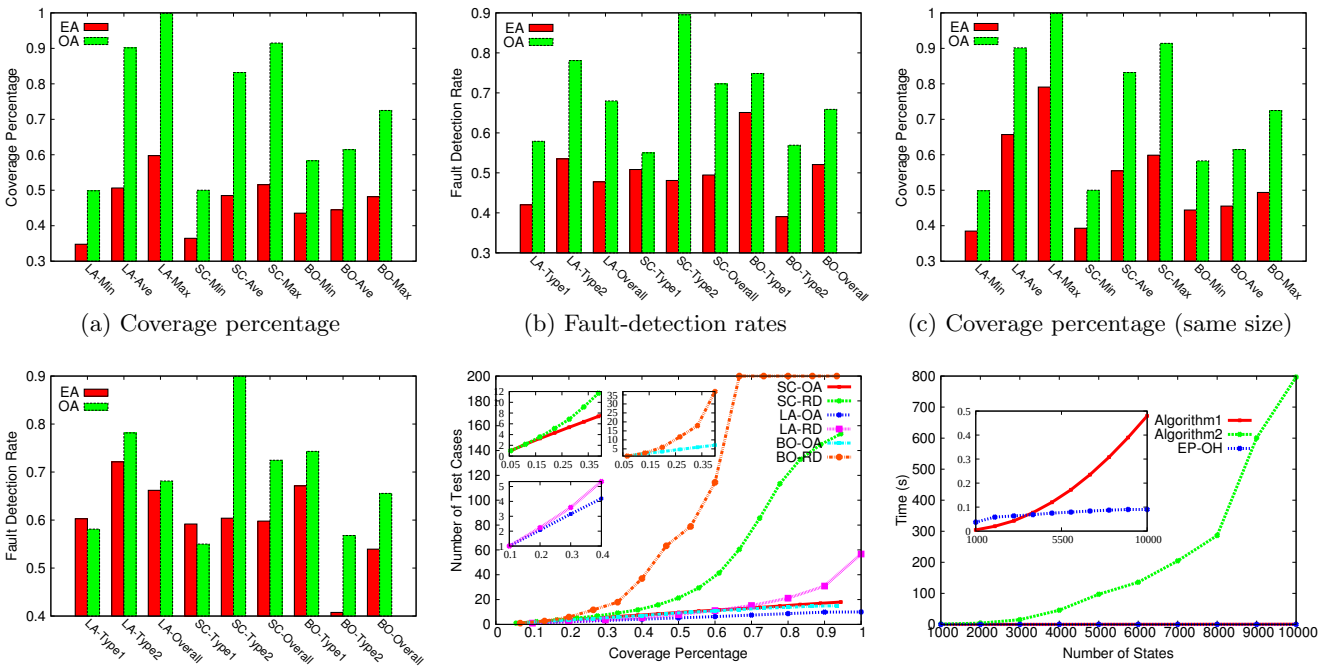
The running time of our algorithms and overhead for event exposure (denoted as EP-OH) are shown in Fig. 5(f). The results show that it takes less than 1 second to derive an event interface using Algorithm 1 and less than 0.1 second to expose all the declared events at runtime for a service with 10,000 states. For a service composition with 10,000 states, Algorithm 2 uses less than 800 seconds to construct all the potentially feasible observations. Therefore, the overhead for both algorithms and event exposure are small.

4.3 Threats to Validity

The validity of the experimental results may be threatened in the following ways:

Construct validity. The experimental results may be invalid if concepts were mismeasured using wrong variables. One purpose of our experiments is to evaluate the benefits of our approach, which include accurate coverage reasoning, more adequate testing of a service composition, and effectiveness of test case generation. Therefore, we measured the quantitative benefits of our approach in terms of coverage

the existing work is equivalent to the average percentage of the path coverage rate reported by all the involved services.



(a) Coverage percentage (b) Fault-detection rates (c) Coverage percentage (same size)

(d) Fault-detection rates (same size) (e) Number of test cases (f) Time complexity

Figure 5: Coverage percentage, fault-detection rate, number of test cases, and time complexity.

rate, fault-detection rate and number of test cases needed.

Internal validity. Confounding factors like the types of seeded faults and test case selections may affect the cause-effect relationships in the experiments if the seeded faults are sensitive to only particular paths across the service composition or the test cases are selected to cover each service well but only cover a few paths in the service composition as a whole. We alleviate the impacts of these factors by seeding different types of faults evenly across a service composition following the guidelines from [12] and randomly selecting test cases from a large pool of test cases.

External validity. To make sure that the experiments can be generalized, we use three representative applications in the experiments because few real-life service composition applications are publicly available. Such applications are also used in existing service testing work [2, 14, 20, 30].

Theoretical reliability. Finally, we repeated the experiments many times to remove accidental errors.

5. DISCUSSION

Deriving the formal model. In this paper, we model a service as a finite state machine. In practice, services may be implemented in BPEL and other languages. We can apply many existing work to transform BPEL services into formal models, such as finite state machines [11] and process algebras [10] to just name a few. On the other hand, to derive the causality conditions among events, the semantics of actions are needed. Some existing Web service standards (like OWL-S [27]) provide such semantics for services (e.g., the pre/post conditions). In addition, the formal models of services with the semantics of actions can be derived using symbolic execution techniques [7]. Our work can be applied to the model based on the existing work.

Event generation and propagation. In our work, services need to generate and propagate events to service consumers during testing. We make no assumptions on how services do that. In practice, aspect-oriented programming techniques can be used to generate events in a way transparent to the service implementations. Events can be propagated to service consumers using a pub/sub middleware [15], or using existing standards like WS-Eventing [27].

Asynchronous communication. To ease the presentation and illustration of our approach, we assume services communicate with each other using synchronous communication. Our approach is also applicable to asynchronous communication. To support asynchronous communication, queues can be introduced in Definitions 5 and 6 to buffer the asynchronous messages from partners. A feasible execution and a feasible observation can be derived in the same way.

Privacy concern. In our work, only necessary events are exposed to abstract and reveal coverage-related internal state changes inside a service. All other state changes inside a service and how states are changed (i.e., by what tasks in the business logic) remain invisible to service consumers. In this way, the privacy concern of service providers is respected to a large extent. On the other hand, sometimes the causality conditions for events may be related to business interests (e.g., the decision making strategies to choose different execution paths inside a service etc) so that service providers may not be willing to expose them. An alternative solution is that each service provider can derive the final conditions that must be satisfied for its own service along each given feasible observation, and provide it to service consumers to avoid revealing individual decision making strategies. This collaborative solution can be applied when the causality conditions are unavailable from the service formal model. We will explore in this direction in our future work.

Constraint solving. Our approach applies constraint

solving techniques to derive test cases for given feasible observations. Constraints may not always be solvable. In this case, our approach can still be applied to determine test coverage using the test cases generated by existing work (e.g., random testing [19]). An alternative approach is to apply the aforementioned collaborative approach to generate test cases for service compositions.

Parallel events. In practice, a BPEL process may involve concurrent executions (e.g., flows), which may generate events to interleave with each other in many ways. The finite state machine model can describe all the possible interleavings as different paths. Service providers can also choose to keep some combinations and remove the others in the event interface to reduce the number of paths (since these different combinations corresponding to the same concurrent execution paths) based on the testing requirements (e.g., examining every possible interleaving is needed in some critical requirements).

Composite Web services. In a service composition, the involved services may be composed of other services. The coverage-equivalent event interface of a composite service should be derived based on the event interfaces of its composed services. This requires to aggregate the events from the involved services into high-level events and construct their causality relationships and conditions. We will explore this in our future work.

6. CONCLUSIONS

White-box testing of service compositions is difficult because service providers usually hide the service implementation details due to business interests or privacy concerns. This paper presents a novel approach to white-box test service compositions based on event exposure from Web services. By deriving coverage-equivalent event interfaces from service implementations, events are defined and exposed from services to accurately determine the test coverage of a service composition at runtime. In this way, service consumers can gain confidence on how adequately a service composition has been tested. An approach to effectively design test cases based on event interfaces is also proposed and the correctness of the approach is proven. Algorithms are developed to derive coverage-equivalent event interfaces and construct feasible observations. The experimental results show that our approach outperforms existing approaches in terms of coverage rate, fault-detection rate and effectiveness of test case generation.

7. REFERENCES

[1] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti. Whitening soa testing. In *ESEC/FSE '09*, pages 161–170, 2009.

[2] L. Bentakouk, P. Poizat, and F. Zaïdi. A formal framework for service orchestration testing based on symbolic transition systems. In *TESTCOM '09/FATES '09*, pages 16–32, 2009.

[3] A. Bertolino, G. Angelis, L. Frantzen, and A. Polini. Model-based generation of testbeds for web services. In *TestCom '08 / FATES '08*, pages 266–282, 2008.

[4] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *WWW '05*, pages 148–159, 2005.

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

[6] K. Chandy. Event-driven applications: costs, benefits and design approaches. In *Gartner Application Integration and Web Services Summit 2006*, 2006.

[7] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9*, pages 109–120, 2001.

[9] M. Emmi, D. Giannakopoulou, and C. S. Păsăreanu. Assume-guarantee verification for interface automata. In *FM '08*, pages 116–131, 2008.

[10] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE '03*, pages 152–161, 2003.

[11] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04*, pages 621–630, 2004.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94*, pages 191–200, 1994.

[13] IBM. Loan approval. http://www.ibm.com/developerworks/webservices/library/ws-bpelcol5/.

[14] K. Kaschner and N. Lohmann. Automatic test case generation for interacting services. pages 66–78, 2009.

[15] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web*, 4(1):1–33, 2010.

[16] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. Bpel4ws unit testing: Framework and implementation. In *ICWS '05*, pages 103–110, 2005.

[17] Z. J. Li, H. F. Tan, H. H. Liu, J. Zhu, and N. M. Mitsumori. Business-process-driven gray-box soa testing. *IBM Syst. J.*, 47(3):457–472, 2008.

[18] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.

[19] J. Mayer and C. Schneckenburger. An empirical analysis and comparison of random testing techniques. In *ISESE '06*, pages 105–114, 2006.

[20] L. Mei, W. Chan, and T. Tse. Data flow testing of service-oriented workflow applications. In *ICSE '08*, pages 371–380, 2008.

[21] L. Mei, W. K. Chan, and T. H. Tse. Data flow testing of service choreography. In *ESEC/FSE '09*, pages 151–160, 2009.

[22] L. Mei, W. K. Chan, T. H. Tse, and R. G. Merkel. Tag-based techniques for black-box test case prioritization for service testing. In *QSIC '09*, pages 21–30, 2009.

[23] R. D. Nicola and M. Hennessy. Testing equivalence for processes. In *ICALP'83*, pages 548–560, 1983.

[24] Oracle. Book ordering. http://www.oracle.com/technology/sample_code/products/bpel/index.html.

[25] OSOA. Sca event processing. http://www.osoa.org/.

[26] J. Tretmans. Test generation with inputs, outputs, and quiescence. In *TACAs '96*, pages 127–146, 1996.

[27] W3C. Ws-eventing, owl-s. http://www.w3.org.

[28] WS-I. Supply chain management. http://www.ws-i.org/deliverables/workinggroup.aspx?wg=sampleapps.

[29] C. Ye, S. C. Cheung, W. K. Chan, and C. Xu. Atomicity analysis of service composition across organizations. *IEEE Trans. Softw. Eng.*, 35(1):2–28, 2009.

[30] Y. Yuan, Z. Li, and W. Sun. A graph-search based approach to bpel4ws test generation. In *ICSEA '06*, page 14, 2006.

[31] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

APPENDIX

A. PROOF OF THEOREM 1

Part 1: Given a feasible execution $((s_{1,j_1}, w_1), \cdots, (s_{n,j_n}, w_n))$ of P, according to its Definition, there exists a sequence of $s_0 \rightarrow s_{i1} \rightarrow \cdots \rightarrow s_{ik}$, where $s_{ik} \equiv ((s_{1,j_1}, w_1), \cdots, (s_{n,j_n}, w_n))$. We can construct a feasible observation (h_1, h_2, $\cdots$, h_n) for this feasible execution in the following way: In the beginning, that is, in state s_0, $h_i \equiv \{e_{i,0}\}$, where $e_{i,0}$ is the start event of P_i. Suppose an event $e_j \in E_i$ is raised during $s_{il} \rightarrow s_{i(l+1)}$, according to the causality definition, $(tail(h_i), e_j) \in R_i$. Therefore, based on Definition 6, $(h_1, h_2, \cdots, h_n) \Rightarrow (h_1, h_2, \cdots, h_i e_j, \cdots, h_n)$. Hence, during each step of $s_0 \rightarrow s_{i1} \cdots \rightarrow s_{ik}$, if an event defined in an event interface is exposed, we can construct a feasible observation from current observation with this new generated event.

Part2: Given a feasible observation (h_1, h_2, $\cdots$, h_n), according to its Definition, there exists a sequence $(e_{1,0}, e_{2,0}, \cdots, e_{n,0}) \Rightarrow \cdots \Rightarrow (h_1, h_2, \cdots, h_n)$, where $e_{i,0}$ is the start event of P_i. We can construct a feasible execution of P for this feasible observation in the following way: In the beginning, that is, the observation is $(e_{1,0}, e_{2,0}, \cdots, e_{n,0})$, and the service composition is in the initial state s_0. Suppose $(e_{1,0}, e_{2,0}, \cdots, e_{n,0}) \Rightarrow (e_{1,0}, e_{2,0}, \cdots, e_{i,0} e_{i,1}, \cdots, e_{n,0})$, then $(e_{i,0}, e_{i,1}) \in R_i$. According to the Definition of causality, there exists an execution of service P_i, that is, $s_{i,0} \xrightarrow{t_{i,1}} \cdots s_{i,j-1} \xrightarrow{t_{i,j}} s_{i,j}$, and $e_{i,1} \equiv e_{s_{i,j-1} \rightarrow s_{i,j}}$. For each step $s_{i,l-1} \xrightarrow{t_{i,l}} s_{i,l}$, according to Definition 5, we can construct a corresponding feasible execution $s_{l-1} \rightarrow s_l$ for this service composition. Therefore, given a feasible observation $(e_{1,0}, e_{2,0}, \cdots, e_{n,0}) \Rightarrow \cdots \Rightarrow (h_1, h_2, \cdots, h_n)$, there exists a corresponding feasible execution $((s_{1,0}, \{\}), \cdots, (s_{n,0}, \{\})) \rightarrow ((s_{1,j_1}, w_1), \cdots, (s_{n,j_n}, w_n))$.

Based on the Part1 and Part2, the conclusion follows.

B. APPLICATIONS AND SEEDED FAULTS

Table 2: Description of Approval Service

t_1	receive approval request	t_2	make decision
t_3	reject approval	t_4	send result
t_5	calculate approval	t_6	reject approval
g_1	$amount < 9$	g_2	$amount \geq 9$
g_3	$reject = true$	g_4	$reject = false$

Table 3: Description of Risk Assessment Service

t_1	receive assessment request	t_2	check record
t_3	check deposit	t_4	assign low-risk level
t_5	send result	t_6	assign high-risk level
t_7	assign high-risk level	t_8	assign high-risk level
g_1	$amount < 3$	g_2	$amount \geq 3$
g_3	$hasrecord = false$	g_4	$hasrecord = true$
g_5	$hasdeposit = true$	g_6	$hasdeposit = false$

Table 4: Description of Loan Service

t_1	receive loan request	t_2	send riskassess request
t_3	receive riskassess result	t_4	approve loan
t_5	notification	t_6	loan approval request
t_7	loan approval result	t_8	assign approval
t_9	loan approval request		
g_1	$amount < 5$	g_2	$amount \geq 5$
g_3	$risklevel = low$	g_4	$risklevel = high$

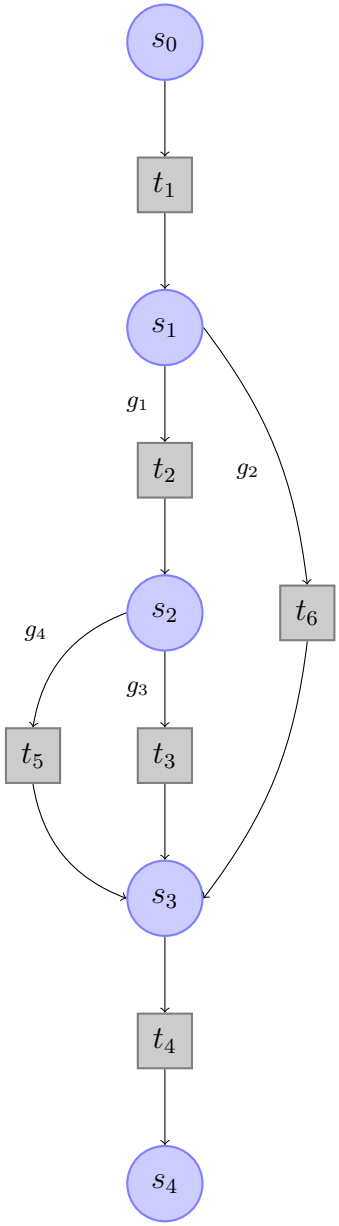


Figure 6: Approval Service

Table 5: Description of Seeding Faults for LoanApproval Application

Faulty Name	Fault Type	Fault Description
f_1	Type 1	Approval: the threshold of g_1 and g_2 are changed to affect the service locally
f_2	Type 2	Approval: g_3 and g_4 are changed to generate inconsistent results
f_3	Type 2	Loan: the threshold of g_1 and g_2 is chanted to generate inconsistent results
f_4	Type 1	Loan: g_1 and g_2 are changed locally
f_5	Type 1	Loan: g_3 and g_4 with additional constraints
f_6	Type 1	Loan: t_8 failure with wrong variable
f_7	Type 2	RiskAssessment: g_3 and g_4 are exchanged to generate inconsistent results
f_8	Type 2	RiskAssessment: g_1 and g_2 have wrong threshold value to generate inconsistent results

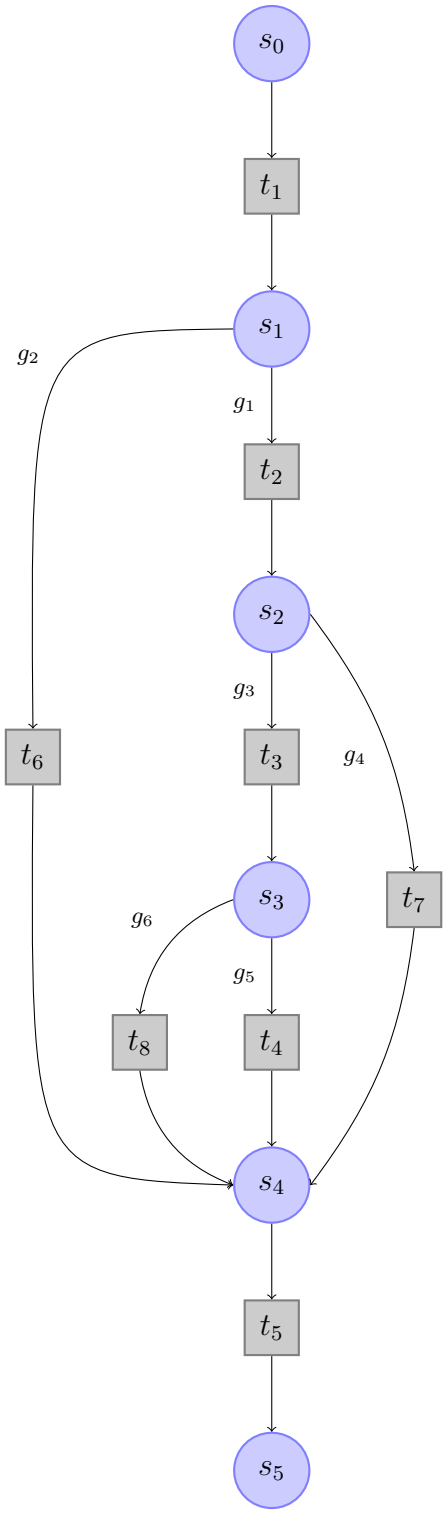


Figure 7: Risk Assessment Service

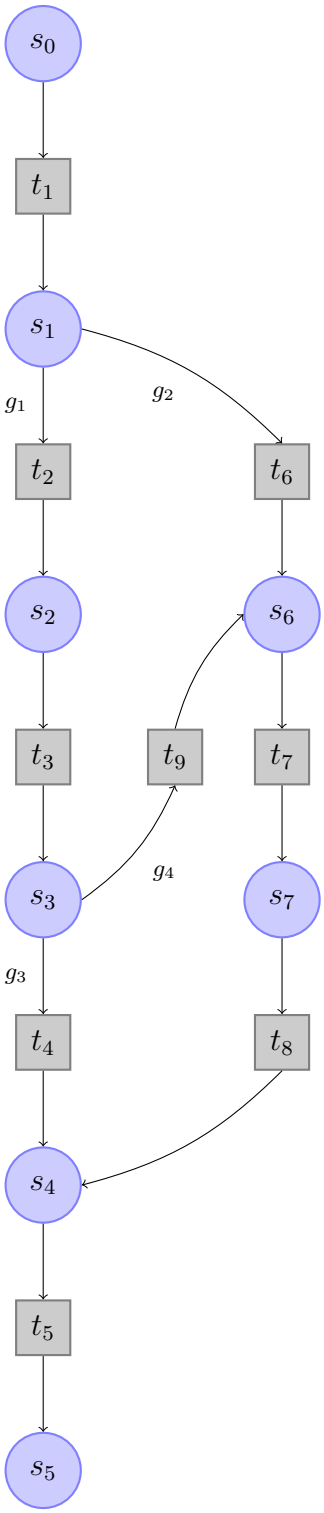


Figure 8: Loan Service

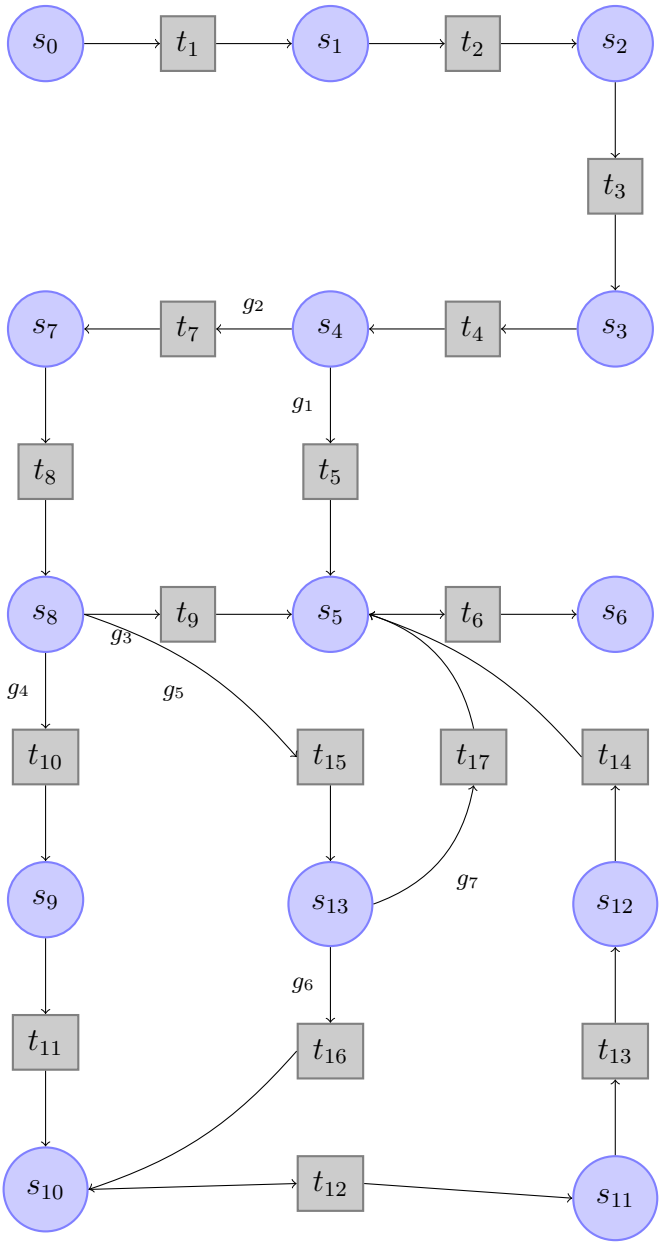


Figure 9: Book Ordering Service

Table 6: Description of Book Ordering Service

t_1	receive customer request	t_2	get customer info
t_3	query customer credit	t_4	check customer credit
t_5	prepare delivery	t_6	deliver result
t_7	payment request	t_8	query credit level
t_9	notify no enough credit	t_{10}	customer confirm
t_{11}	payment preparation	t_{12}	payment transaction
t_{13}	receive approval	t_{14}	prepare delivery
t_{15}	receive new amount	t_{16}	prepare new payment
t_{17}	prepare delivery		
g_1	$acredit\_request \leq 0$	g_2	$credit\_request > 0$
g_3	$credit\_level \leq 0$	g_4	$credit\_level \geq credit\_check\_request$
g_5	$credit\_level < credit\_check\_request$	g_6	$new\_amount \leq credit\_level + account\_credit$ & $new\_amount > account\_credit$
g_7	$new\_amount > credit\_level + account\_credit$ | $new\_amount \leq account\_credit$		

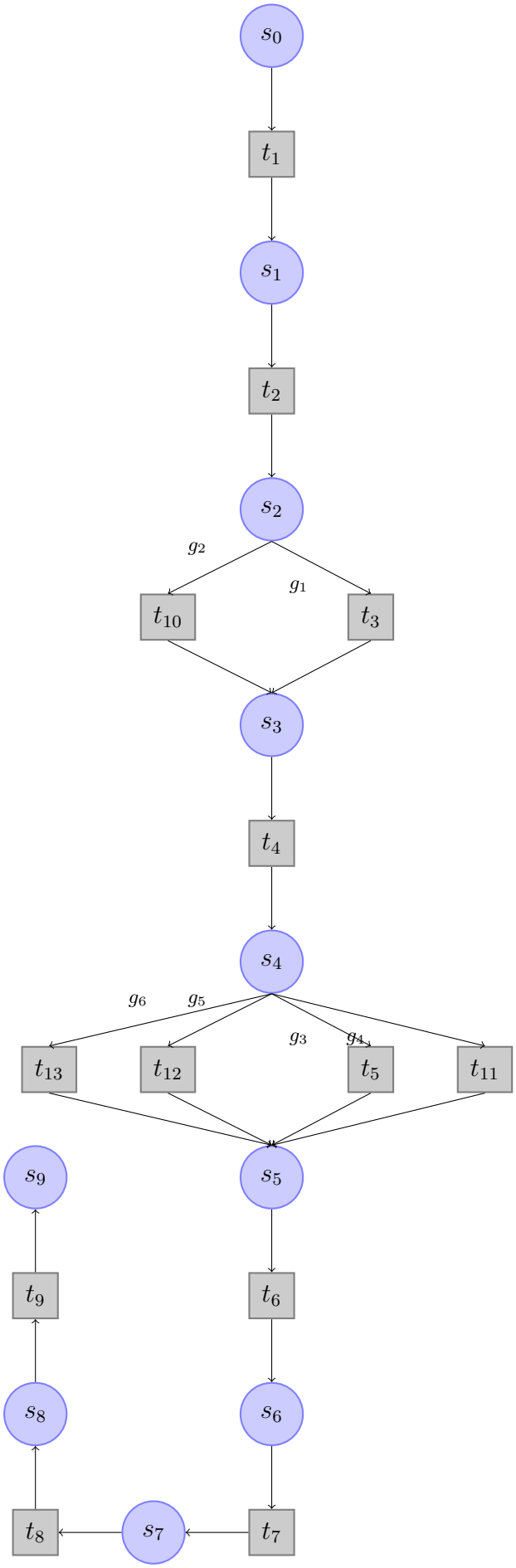


Figure 10: Credit Card Service

Table 7: Description of Credit Card Service

t_1	receive payment request	t_2	check customer credit
t_3	calculate credit promotion	t_4	check interests
t_5	calculate credit	t_6	send credit level
t_7	receive payment	t_8	approve result
t_9	send payment result	t_{10}	calculate credit
t_{11}	calculate credit	t_{12}	calculate credit
t_{13}	calculate credit		
g_1	$customerID < 6$	g_2	$customerID \geq 6$
g_3	$creditcard\_credit \geq credit\_check\_request$ & $hasOneYearContractPlan == false$	g_4	$creditcard\_credit \geq credit\_check\_request$ & $hasOneYearContractPlan == false$
g_5	$creditcard\_credit < credit\_check\_request * 2$&$hasInterests == true$	g_6	$creditcard\_credit \geq credit\_check\_request * 2$&$hasInterests == true$

Table 8: Description of Seeding Faults for BookOrdering Application

Faulty Name	Fault Type	Fault Description
f_1	Type 1	Credit Card: t_3 generates wrong result
f_2	Type 2	Credit Card: t_{10} generates inconsistent result
f_3	Type 2	Credit Card: t_{12} generates inconsistent result
f_4	Type 1	Credit Card: g_5 and g_6 with additional constraints
f_5	Type 2	Credit Card: t_5 generates inconsistent result
f_6	Type 2	Book Ordering: g_1 and g_2 with additional constraints to generate inconsistent decisions
f_7	Type 1	Book Ordering: g_3 with additional constraints
f_8	Type 2	Book Ordering: g_4 and g_5 with wrong constraints to generate inconsistent results
f_9	Type 1	Book Ordering: t_{11} generates failure for particular customers
f_{10}	Type 1	Book Ordering: t_{16} generate side effects for particular customers

Table 9: Description of Manufacturer Service

t_1	receive manufacturing request	t_2	query stock
t_3	complementation	t_4	sending result
t_5	query factoryA	t_6	factoryA Produce request
t_7	query factoryB	t_8	factoryB Produce
t_9	query factoryC	t_{10}	factoryC Produce
t_{11}	complete assign		
g_1	$manufacturer\_stock \geq requested$	g_2	$requested > manufacturer\_stock$
g_3	$requested \leq manufacturer\_stock + factoryA\_can\_produce$	g_4	$requested > manufacturer\_stock + factoryA\_can\_produce$
g_5	$requested \leq manufacturer\_stock + factoryA\_can\_produce + factoryB\_can\_produce$	g_6	$requested > manufacturer\_stock + factoryA\_can\_produce + factoryB\_can\_produce$
g_7	$requested \leq manufacturer\_stock + factoryA\_can\_produce + factoryB\_can\_produce + factoryC\_can\_produce$	g_8	$requested > manufacturer\_stock + factoryA\_can\_produce + factoryB\_can\_produce + factoryC\_can\_produce$

Table 10: Description of Retailer Service

t_1	receive customer request	t_2	query warehouseA
t_3	transaction for A	t_4	deliver product
t_5	prepare_complement_warehouseA	t_6	Complement warehouse
t_7	wait for manufacturer	t_8	assign complemented
t_9	query warehouseB	t_{10}	transaction for B
t_{11}	prepare_complement_warehouseB	t_{12}	query warehouseC
t_{13}	transaction for C	t_{14}	prepare_complement_warehouseC
g_1	$product\_type == 0$	g_2	$product\_type == 1$
g_3	$product\_type == 2$	g_4	$stockA \geq amount$
g_5	$stockA < amount$	g_6	$stockB \geq amount$
g_7	$rstockB < amount$	g_8	$stockC \geq amount$
g_9	$stockC < amount$		

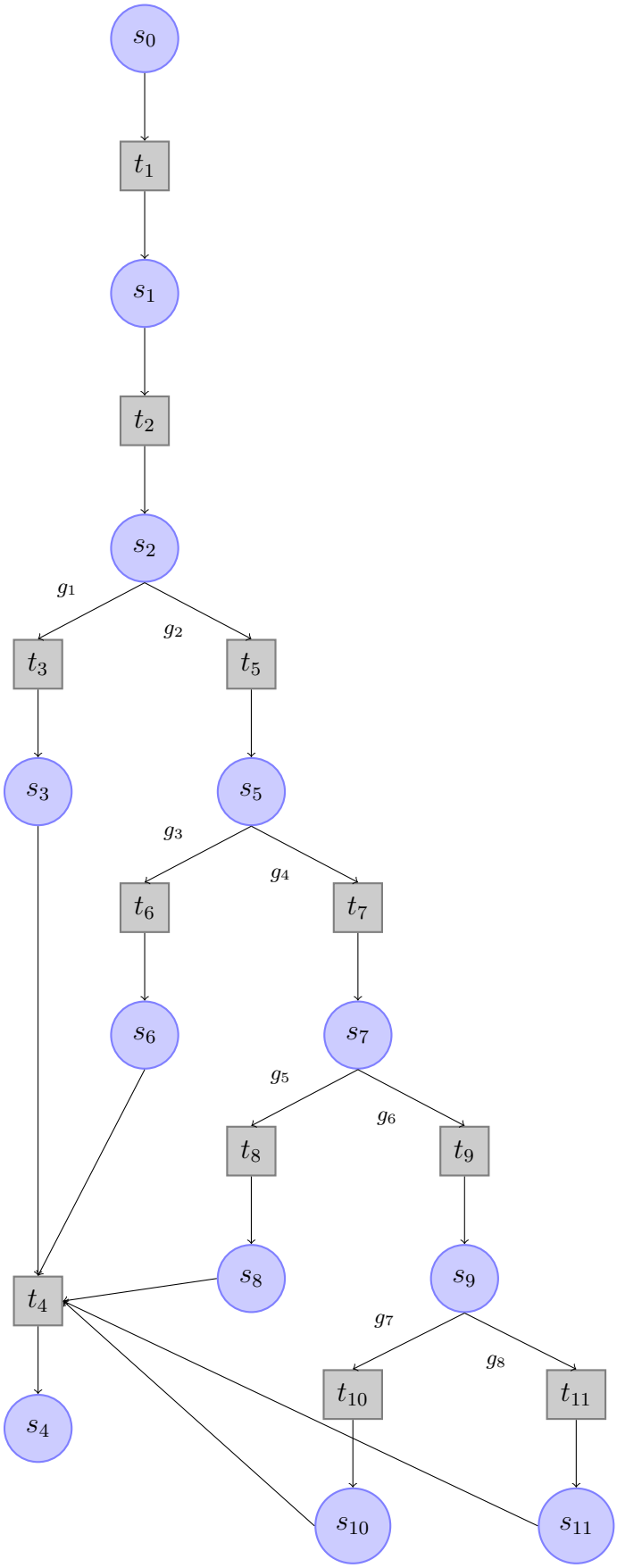


Figure 11: Manufacturer Service

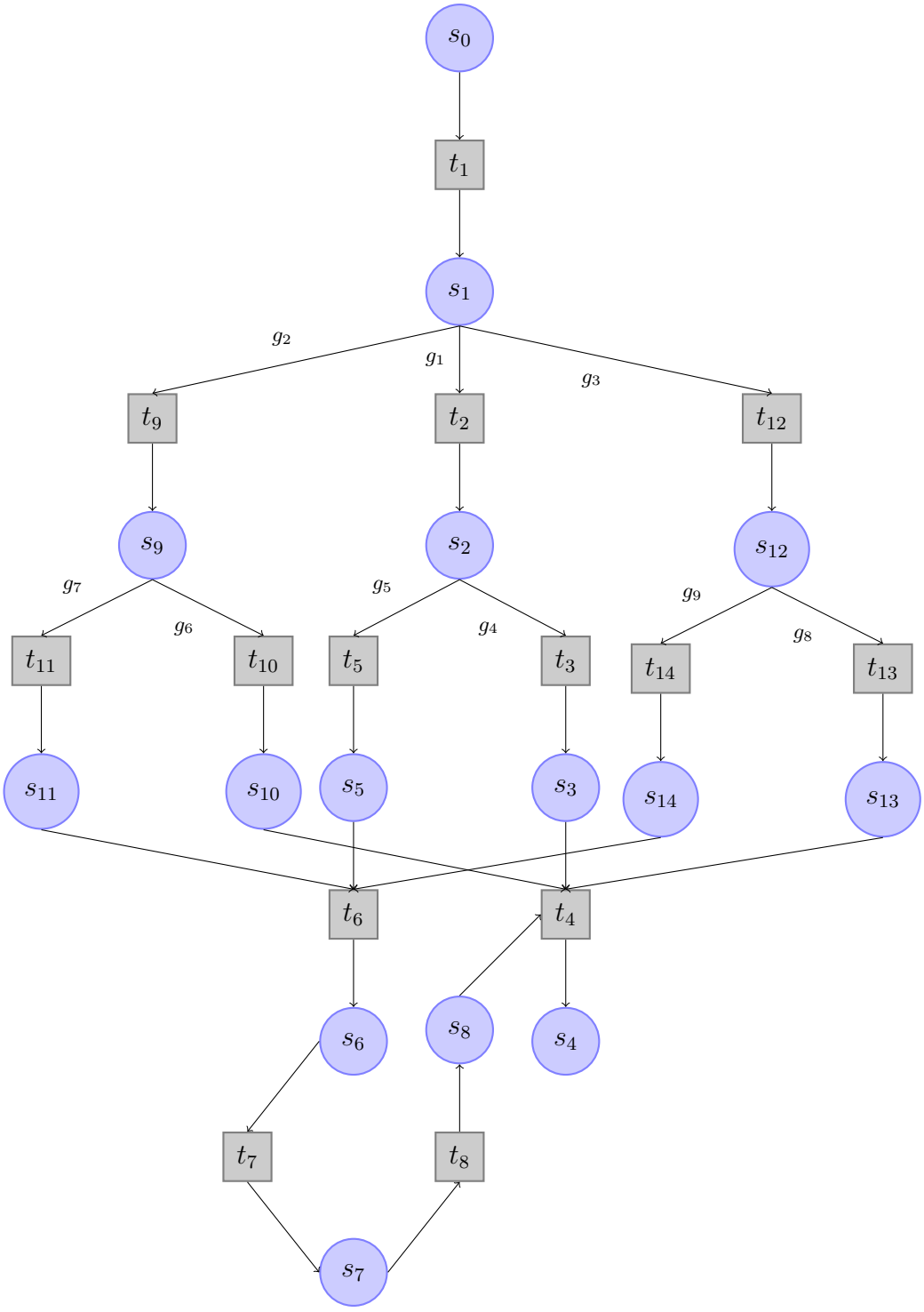


Figure 12: Retailer Service

Table 11: Description of Seeding Faults for SupplyChain Application

Faulty Name	Fault Type	Fault Description
f_1	Type 1	Manufacturer: g_1 and g_2 are exchanged
f_2	Type 1	Manufacturer: g_3 and g_4 are exchanged
f_3	Type 1	Manufacturer: g_7 and g_8 are exchanged
f_4	Type 2	Manufacturer: t_3 generates inconsistent result
f_5	Type 2	Manufacturer: t_6 generates inconsistent result
f_6	Type 1	Manufacturer: t_8 generates wrong result
f_7	Type 1	Retailer: the threshold of g_1 is changed
f_8	Type 1	Retailer: g_8 and g_9 have additional constraints
f_9	Type 2	Retailer: g_6 and g_7 with wrong variable to generate inconsistent decisions
f_{10}	Type 2	Retailer: t_5 calculates wrongly for some range of input
f_{11}	Type 2	Retailer: t_{11} calculates wrongly for some range of input
f_{12}	Type 2	Retailer: t_{14} calculates wrongly for some range of input