Sifter: A Service Isolation Strategy for Internet Applications

Chunyang Ye, S.C. Cheung, and W.K. Chan

Abstract—Service oriented architecture (SOA) provides a flexible platform to build collaborative Internet applications by composing existing self-contained and autonomous services. However, the implicit interactions among the concurrently provisioned services may introduce interference to Internet applications and cause them to behave abnormally. It is thus desirable to isolate services to safeguard their application consistency. Existing approaches mostly address this problem by restricting concurrent execution of services to avoid all the implicit interactions. These approaches, however, compromise the performance and flexibility of Internet applications due to the long running nature of services. This paper presents Sifter, a new service isolation strategy for Internet applications. We devise in this strategy a novel static approach to analyze the potential implicit interactions among the services and their impacts on the consistency of the associated Internet applications. By locating only those afflicted implicit interactions that may violate the application consistency, a novel approach based on exception handling and behavior constraints is customized to involved services to eliminate their impacts. We show that this approach exempts the consistency property of Internet applications from being interfered at runtime. The experimental results show that our approach has a better performance than existing solutions.

Index Terms—Atomicity sphere, behavior constraint, exception handling, implicit interaction, isolation, web service.

1 INTRODUCTION

Service oriented architecture (SOA) provides a flexible paradigm to build collaborative Internet applications in the cloud environments. In this paradigm, service providers develop reusable components and publish them as services. Service consumers can select suitable third-party services and compose them to develop their own Internet applications.

Since a service provider may provide more than one service, and each service may have multiple instances to serve different service consumers at the same time, how to avoid interference among different services (or instances) becomes a crucial issue in the lifecycle of each service. For example, we found that services can implicitly interact with one another when their underlying tasks compete for shared resources. Such implicit interactions may introduce interference between services and cause them to behave unintendedly and unacceptable to service consumers at runtime. In this paper, we study the problem of interference among services caused by their interactions beyond message exchanges and their impacts on the application consistency, a desirable criterion to evaluate whether services function correctly to handle errors and exceptions [9]. To guarantee application consistency, services are usually required to be atomic, in the sense that a service can either complete an expected business transaction successfully or cancel it, without any side effects.

Let us illustrate the problem using an example adapted from [24], where there are two service providers, a manufacturer and a workshop, as illustrated in Fig. 1. The manufacturer provides two services $s_1$ and $s_2$, where $s_1$ is to design a product, and $s_2$ is to manufacture the product. The workshop provides a service $s_3$ to manufacture some customizable items for $s_2$. To improve the efficiency, the manufacturer applies the pipeline techniques to design products and produce them at the same time. This is reasonable because the testing of a product design may take a long time, but the historical faulty rate for product design is low. Therefore, $s_1$ interacts with $s_2$ implicitly by sharing the design documentation in the public database (that is, "$s_2$: Write BOM", and "$b_1$: Read BOM"). $s_2$ may compose different services from different workshops to produce different customizable items. Nevertheless, if the testing reveals any problems in the design of a product, $s_1$ needs to fix the problems and update the design accordingly. As a result, $s_2$ should stop the current manufacturing and use the updated design to manufacture the products. However, if the workshop service $s_3$ used by $s_2$ has committed certain task whose effects are unable to compensate, the atomicity is violated. For example, the task "$c_2$: produce items" may have been committed and its effects are unable to compensate, because the produced items could not be used by others. As a result, the produced items become a waste, but they have consumed resources (e.g., parts and materials).
Hence, the manufacturer may have to pay extra cost for these wrong products and endure the value loss.

Note that in service-based systems, the implicit interactions may be far more complex than above example and introduce unexpected interference to services due to their specific data integrity requirements for the shared resources (cf. Section 3.3). Moreover, such interference may be introduced from time to time when services evolve or new services are deployed. The cloud environment further exaggerates such interference, imposing great challenges to the development, deployment and maintenance of services for Internet applications. It is therefore desirable to isolate services from the interference caused by such interactions to assure their integrity.

Since the interference among different services is introduced by their concurrent accesses to the shared resources, a simple solution is to forbid any concurrent execution for certain combinations of services. This approach should be taken as the last resort because it may seriously compromise the performance (e.g., the degree of concurrency) of the legitimate service compositions among service customers and service providers, as service transactions may however last for weeks or even months. Conventional solutions in conventional database transactions [24] (such as isolating services by scheduling them in a serializable way with the aid of proper locking) are also inadequate due to similar reasons.

Another solution is to devise a sandbox (e.g., docker [17]) to isolate services by separating the resources they use and wrap them into independent containers. Such a solution is desirable in some development style such as DevOps [16] because wrapping the services and resources into containers makes it easy to deploy and migrate the applications to different dynamic environments. However, this solution not only compromises the resource usability for long running services but also is not applicable to some scenarios, especially when the sharing of resources among services is needed (cf. Fig. 1).

In this paper, we propose a novel fine-grained service isolation strategy, namely Sifter, to trade off above two extreme solutions concerning the balance of service performance and resource usability. By locating the potential implicit interactions that would interfere the atomicity in each application, our solution accurately eliminates the potential inconsistency scenarios with the aid of exception handling and behavior constraints. Such a solution customizes the isolation of each service in its different applications, and handles the implicit interactions in each application separately. As a result, services are isolated in a fine-grained way to work correctly to preserve their application consistency, and both their performance and resource usability are respected to a large extent.

Note that the idea of developing a fine-grained solution to isolate services was first explored in our preliminary work [36], but only two types of implicit interactions were investigated. Our further investigation shows that the implicit interactions are far beyond these two types and more general (cf. Section 3.3). In this paper, we continue to explore this idea to cope with more general implicit interactions. The challenge lies in how to locate the general implicit interactions and how to analyze the joint impacts of both the implicit and explicit interactions, since the crosscutting of general implicit and explicit interactions among services can lead to more unexpected abnormal behavior than the two fixed patterns. This can also increase the complexity and impose new challenge for the efficient and customizable resolution of such interferences among services.

This paper addresses above challenges with fourfold contributions. First, we investigate the implicit interactions among services and their side effects in more general scenarios. Second, we propose a novel model to generalize and integrate both the explicit and implicit interactions among services. Based on the model, algorithms are designed to locate the general potential inconsistency scenarios. Third, we propose a fine-grained service isolation strategy to preserve the application consistency of services with the aid of exception handling and behavior constraints. By allowing service providers to customize the isolation of each service in its different applications, this strategy concerns the balance of service performance and resource usability. Fourth, we conduct extensive experiments to evaluate the effectiveness of the service isolation strategy. The experimental results show that our solution outperforms the baseline solutions in terms of preserving the application consistency with a smallest performance overhead.

The rest of this paper is organized as follows: Section 2 presents the preliminary on the service model and the atomicity requirements. Section 3 presents our approach to identifying the afflicted implicit interactions and suppressing their effects on the atomicity property based on behavior constraints and exception handling. Section 4 evaluates our approach with a series of experiments. Section 5 reviews related works and makes a comparison to ours.
In service-oriented architecture, the collaboration between different services is usually referred to as service compositions. In a service composition, services from different service providers communicate with each other through message exchanges between their ports (defined in the service interfaces, such as WSDL [31]). Such communication based on message exchanges is also referred to as explicit interaction. Fig. 2 illustrates the concept model of a service composition. For the sake of behavioral analysis, we model a service by its back-end process.

In this model, services are composed by linking the ports (a link represents a message exchange between the ports) between service interfaces. The communication between services is guided by a service composition specification (e.g., WS-CDL [30]), which defines how messages are exchanged between services, including the message types, the message orders and so on. In addition, the specification may also define how to handle exceptional cases in a service composition when errors occur during the collaboration. For example, if task $a_3$ in Fig. 2 fails to communicate with $b_3$, an exception will be thrown. An exception handler is defined to handle this exceptional case (i.e., using $a'_3$ to communicate with $b'_3$ instead).

We assume services adopt the replacement exception model [38] (which is also adopted in existing service definition languages, such as BPEL [19]). In the replacement exception model, exception handlers semantically replace the failed task (or a set of tasks) and resume the execution of services from the point after the failed task (or the set of tasks). For example, if $a_3$ in Fig. 2 fails to communicate with $b_3$, service $s_1$ stops the execution of tasks in the scope (represented by the dashed oval) and replaces them by the tasks in the exception handler, i.e., the internal task (non-port task) and task $a'_3$ in the dashed rectangle. Then, service $s_1$ resumes the execution from the point after the scope if tasks in the exception handler are committed.

### 2.2 Service Definitions

**Definition 1** (State). A state is defined as a finite set $\{ (x_1, t_1, v_1), \ldots, (x_n, t_n, v_n) \}$, where $x_i$ is a variable, $t_i$ and $v_i$ are its type and value, respectively.

**Definition 2** (Process). A process is defined by a 5-tuple $(p, P, A, G, F_p)$, where $P$ is a set of states, $p \in P$ is the initial state of the process, $A$ is a set of actions, $G$ is the set of guarded boolean expressions, and $F_p \subseteq (P \times G \times A \times P)$ is a quaternary transition relation. For simplicity, we denote a process using its initial state (that is, $p$).

Given any process $p$, if $\exists (p, g, a, q) \in F_p$, and $g$ is satisfied in state $p$ (denoted as $p \models g$), then $p$ can transmit to state $q$ via committing action $a$, denoted as $p \xrightarrow{a} q$. We denote a guarded boolean expression always being “true” as $\top$, and always being “false” as $\bot$. Without loss of generality, we represent a successful termination state of a process as the constant $\true$ and the termination by deadlock as $0$. We assume that each process should terminate [24].

**Definition 3** (Operator). Processes can be manipulated and composed through algebraic operators in the following ways (separated by “|”):

$$p = 0 | \true | (g \triangleright a) \cdot p | p \parallel H | p + p | p : p$$

where $a \in A$, $g \in G$, and $g \triangleright a$ represents $g$ is the precondition to execute $a$; “\parallel”, “\|$” represent the prefix operator, parallel composition operator (and $H \subseteq A$ is the set of synchronization port actions), choice composition operator, sequential composition operator, respectively. “|” is the scope operator, and “\{ $e_i $, $p_i $\}” represents the sequence of exception handlers of a process (i.e., $e_i$ is an exception, and $p_i$ is an exception handler)\(^\dagger\).

In the formal model, an action could be the commitment of a task, which occurs instantaneously, or an exception raised by the execution of a task. Therefore, a task is represented as $a_c \cdot \sqrt{e_1} + \cdots + e_i \cdot \sqrt{t}$, where $a_c$ is the commitment of the task, and $e_1, \ldots, e_i$ represent the possible exceptions raised during the execution of the task\(^\dagger\). Let $E$ represent the set of all exceptions, $E \subseteq A$. Exceptions can be defined hierarchically based on inheritance (like exception types in java). We denote by $e_j \leq e_j$ if $e_j$ has a same type as $e_i$ or $e_j$ inherits $e_i$. In our formal model, we denote $e_0$ as the exception that is the super type of all the exceptions, that is, $\forall e_i \in E, e_0 \leq e_i$.

Actions can be classified as port actions and non-port actions. Processes communicate explicitly with one another via port actions (referred to as an explicit interaction). Non-port actions do not participate in any explicit interaction (exceptions are also non-port actions). We assume that processes communicate with

---

1. The precedence of these operators in descending order is “\{ $e_i $, $p_i $\}”, “|”, “\|$”, “\parallel”, “\triangleright”.
2. For ease of presentation, we use an exception to represent the action of raising the exception.
each other synchronously at the application level (although the underlying communication protocols can be asynchronous). For ease of presentation, our formal model assumes that port actions with the same name from two processes communicate with each other. We also introduce silent actions $\tau_e$ to represent an internal state transition due to an empty action (e.g., empty activity in BPEL [19]) and $\tau_e$ to represent an exception handling (for exception $e$). These two silent actions do not engage in any synchronization, and have no physical meanings and side effects.

The operational semantics of these algebraic operators are explained as follows. A process $p = (q \triangleright e) \cdot q$ represents that after committing task $a$, $p$ reaches state $q$. If $p$ is equal to $q$, that is, $p = (q \triangleright e) \cdot p$, then $p$ is a recursive process (i.e., a loop). $p = (q_1 \triangleright e) \cdot q_1 + (q_2 \triangleright e) \cdot q_2$ represents that if committing task $a$, $p$ reaches state $q_1$; or committing task $b$, $p$ reaches state $q_2$. For two processes $p = (q_1 \triangleright e) \cdot p'$ and $q = (q_2 \triangleright e) \cdot q'$, if $a$ is not a port action, then $p \parallel q$ could commit $a$ and then reach state $p' \parallel q$. Similarly, if $b$ is not a port action, then $p \parallel q$ could commit $b$ and then reach state $p \parallel q'$. If both $a$ and $b$ are not port actions between $p$ and $q$, then $p \parallel q$ could commit either $a$ or $b$ first (suppose their conditions $g_1$ and $g_2$ are satisfied). Suppose both $a$ and $b$ are port actions between $p$ and $q$, if $a = b$ and $g_1$ and $g_2$ are satisfied, then $p \parallel q$ communicates synchronously via the port actions $a$ and $b$, and reaches state $p' \parallel q$; otherwise $p \parallel q$ reaches a deadlock (i.e., 0). If both $p$ and $q$ are $\\sqrt{\ \ }$, then $p \parallel q$ can terminate successfully; otherwise, if either of them is a deadlock 0, $p \parallel q$ will reach a deadlock. Given two processes $p$ and $q$, $p; q$ will first execute $p$ and then $q$ if $p$ is terminated successfully; otherwise, if $p$ reaches a deadlock, then $q$ is not executed. If $p$ raises an exception $e$, then the execution of $p$ is terminated. If exception handlers $(e_j, h_j)$ are defined to catch the exception (that is, $p^{(e_j, h_j)}_1$ is defined and $\exists j \leq e$, then the first matching exception handler is invoked to handle the exception, and the exception $e$ is deactivated into a specific silent action $\tau_e$ (note that $\tau_e \notin E$); otherwise, the exception is propagated to the outside scope (i.e., $p^{(e_j, h_j)}_2, g, e, \sqrt{\ \ } \in p^{(e_j, h_j)}$).

**Definition 4 (Trace).** Given a process $p$, and a sequence of actions $a_1, a_2, \ldots, a_n \in A$, if $s_1, s_2, \ldots, s_n \in P$, such that $a_1 \rightarrow s_1, a_2 \rightarrow s_2, \ldots, a_n \rightarrow s_n$, then the sequence of actions $a_1, a_2, \ldots, a_n$ is called a trace of process $p$ (also called an instance). If $s_n \in \{0, \sqrt{\ \ }\}$, then we call it a complete trace. We denote by $\text{trace}(p)$ the set of all traces of process $p$. For a trace $t$, $t[i]$ denotes the action committed in the $i$th position of $t$ ($i \geq 0$).

**2.3 Atomicity Semantics**

According to the operational semantics of the service model, given a process $p^{(e_j, h_j)}_1$, if an execution of $p$, say $t$, raises an exception $e$, then $t$ is terminated (that is, $t$ is a complete trace of $p$, and $e = t[k]$, where $k$ is the number of actions in $t$). If an exception handler $(e_j, h_j)$ is defined to catch $e$ (i.e., $e_j \leq e$), then the handler $h_i$ is invoked to semantically replace $t$; otherwise $e$ is propagated to an outside scope where the process $p^{(e_j, h_j)}_2$ belongs to. As we adopt the replacement exception model, in either case, instance $t$ is aborted. Therefore, to keep application consistency, all the already committed tasks in $t$ should be compensated. If any of such committed task $a$ is non-compensable (denoted as $NC(a)$ is true), then $t$ violates the atomicity property.

The atomicity property of a service may be violated if the service does not form an atomicity sphere. That is, tasks in the service are organized in a way that its certain execution may need to undo some tasks that are non-compensable. To examine how tasks are organized in a service, we introduce a predicate $reach(a_1, a_2) = true$ to represent that action $a_1$ can reach $a_2$ via a feasible path $(s_0, a_1, b_1, s_1, \ldots, s_{m-1}, b_m, s_m) \in F_{p_k}$, where $p_k$ is a process embedded into $p$ based on Definition 3 ($p_k$ can be equal to $p$), and $a_1 = b_1$, $a_2 = b_m$. For example, suppose $p = (g_1 \triangleright a_1) \cdot p_1$, $p_1 = (g_2 \triangleright a_2) \cdot p_2$, $p_2 = (g_3 \triangleright a_3) \cdot p_3$, then $reach(a_1, a_3) = true$ because there exists a path $(p_1, g_2, a_2, p_2, p_2, g_3, a_3, p_3) \in F_{p_1}$.

Similarly, we can verify that $reach(a_1, a_2) = true$ and $reach(a_1, a_2) = true$. In another example, $p = (g_1 \triangleright a_1) \cdot p_1 \bigcup (s_1, g_2, a_2, p_2, p_2, g_3, a_3, p_3) \in F_{p_k}$, we have $reach(a_2, a_2) = true$ and $reach(a_1, a_2) = true$ but $reach(a_1, a_2) = false$ because $a_1$ is caught by the exception handler $\bigcup (s_1, a_1)$ and transferred to $\tau_e$, for process $p$ (and thus $reach(a_1, \tau_e) = true$).

**Definition 5 (Atomicity sphere).** A process $p$ satisfies the atomicity sphere, denoted as $\varphi(p)$, if and only if the following conditions are satisfied:

1. $\forall(p, g_1, a_1, p_1) \in F_{p_k}, (p_1, g_2, a_2, p_2, p_2, g_3, a_3, p_3) \in F_{p_k},$ if $\forall a_i \notin E$ (i = 1...n - 1) and $a_n \equiv \tau_e$, then $reach(a_1, a_1) = false \lor NC(a_1) = false$.

2. $\forall(p, g_1, a_1, p_1) \in F_{p_k}, (p_1, g_2, a_2, p_2, p_2, g_3, a_3, p_3) \in F_{p_k},$ if $\forall a_i \notin E$ (i = 1...n - 1) and $a_n \in E$, then $NC(a_1) = false$.

Condition 1 represents that if an exception is raised and captured by an exception handler for some process, then all the tasks committed by this process (including its embedded processes) should be compensable. Condition 2 means that if the raised execution is not captured by any pre-defined exception handler, then all the committed tasks should be compensable. For ease of presentation, we denote the set of the non-compensable tasks in process $p$ that do not satisfy Conditions 1 and 2 as $S_{violations}(p)$.

**Definition 6 (Atomicity-equivalent public view).** Let $pv$ be a process with port and silent actions only, $pv$ is an atomicity-equivalent public view of process $p$, if and only
Fig. 3. Implicit interaction (resource conflict).

if for any process \( q, \varphi_{as}(pv \parallel q) = \varphi_{as}(p \parallel q) \).

To check the atomicity sphere criterion in a service composition without disclosing the details of their backend processes, we can derive atomicity-equivalent public views from the backend processes [37]. These public views can be then used to check the atomicity sphere of a service composition instead of using their backend processes.

3 SIFTER

3.1 Motivations and Observations

To illustrate the research issue more generally, let us review another case adapted from [2], where there are two service providers, one supplier and one retailer, as depicted in Fig. 3. The supplier provides multiple services (e.g., \( s_1 \) and \( s_2 \)) to deliver their products to customers. The retailer orders products from the supplier based on its requirement. On receiving an order from the retailer, the supplier first checks whether the required resources (e.g., mobile phone handsets) are available in stock. If the resources are available, the supplier then starts to produce the products based on the retailer’s requirements, and the retailer starts to pre-sell the products. Once receiving the products from the supplier, the retailer delivers them to its customers. In this case, if services \( s_1 \) and \( s_2 \) are executing concurrently, \( s_1 \) may interact with \( s_2 \) implicitly and cause the collaboration between \( s_2 \) and \( s_3 \) to fail. For example, after \( s_2 \) checks the availability of resources in stock, service \( s_1 \) happen to use up all the resources to produce products in some other collaboration, leading to “\( b_1 \).produce” failure for service \( s_2 \). However, service \( s_3 \) may have pre-sold the products to its customers, and breaching the sale contracts will incur nonetheless penalty to the retailer (e.g., loss of reputation and damage of the public image of the retailer). The atomicity property is thus violated.

Note that in service-based systems, the implicit interactions may be far more complex than the two cases in Fig. 1 and Fig. 3 due to their specific data integrity requirements for the shared resources (cf. Section 3.3) and introduce unexpected interference to the services. This suggests us to isolate services from the interference caused by implicit interactions to make them work correctly. We observed that not all implicit interactions will cause atomicity violations in a service composition. For example, if the supplier’s resource in Fig. 3 still remains more than required by service \( s_2 \) after \( s_1 \) uses a small part of them, this implicit interaction will not cause \( s_2 \) fail. As a result, the composition of \( s_2 \) and \( s_3 \) still satisfies the atomicity property. We also observed that an implicit interaction associated with one service may not always cause atomicity violations in every application involving this service. For example, if the manufacture service \( s_2 \) in Fig. 1 composes another workshop service whose task “\( c_2 \).Produce items” is compensable (e.g., the items can be reused), then the atomicity property is not affected by the implicit interaction between the design service and the manufacture service.

These findings inspire us to customize the solutions to eliminate the impacts of implicit interactions in a fine-grained way. That is, instead of concerning all the implicit interactions, we only need to handle those afflicted ones that may affect the application consistency. Moreover, we can customize the handling of an afflicted implicit interaction associated with a service in different applications based on the quality requirement (e.g., the rate of concurrent executions) of each application. In this way, the service can be reused in different applications with different quality requirements. As a result, the resource usability, the performance and reusability of the involved services are respected to a large extent.

3.2 Overview

As explained in Section 1, existing solutions to address the implicit interactions compromise either the performance of services or the efficiency of resource usage and the reusability of services. To address the limitations of these approaches, we present in this section a novel fine-grained service isolation strategy, namely Sifter, for Internet applications. As illustrated in Fig. 4(a), by locating the afflicted implicit interactions associated with each service, Sifter eliminates the impacts of those afflicted implicit interactions only and filter the others. As a result, services are isolated to work correctly (in terms of preserving their atomicity property) with an acceptable performance overhead and resource usage efficiency.

Fig. 4(b) illustrates the two-stage methodology of Sifter. At stage 1, service providers develop services, deploy them, and publish their atomicity information (i.e., atomicity-equivalent public views [35]) to service consumers. At stage 2, service consumers can use such information to select suitable services whose composition satisfies the atomicity sphere [37]. Then, the service consumer and the involved service providers enact the service level agreement (SLA) on the atomicity property. According to the SLA, each involved
service provider analyzes the potential afflicted implicit interactions associated with its services (cf. Section 3.4). Based on the analysis result, each service provider can configure its services by enacting new exception handlers to handle the exceptions raised by afflicted implicit interactions (and this can be done interactively with locating afflicted interactions), or choose to enforce behavior constraints to restrict the usage of shared resources for its other services. Note that whether enacting exception handling or enforcing behavior constraints depends on the quality and functionality requirements from service consumers (cf. Section 3.5).

3.3 Formal Model

To analyze the potential atomicity violations in service compositions caused by implicit interactions and eliminate their impacts, we extend the service model to describe the implicit interactions, as shown in Fig. 4(c). In the model, implicit interactions are more general due to the variety of resource sharing patterns among services, each of which may have its own data integrity requirements for the shared resources. For example, services $s_1$ and $s_2$ access the same shared resource $r_i$ to form an implicit interaction but they have different data integrity requirements. However, if a data integrity requirement of one service instance is violated by another service instance, then the first service instance is interfered by the second one. It is desirable to isolate services from the interference caused by such interactions to assure their integrity. Note that the term “isolation” in this paper is different from the ACID properties in traditional database systems, in the sense that services are not strictly isolated but are allowed to relax the capability of allowing interference among them due to their long running nature, nevertheless their application consistency can be kept. For instance, an exception will be usually thrown and propagated to the first service instance in the aforementioned example (e.g., by the BPEL engine), which may then trigger exception handlers to resolve the data integrity violation to keep its atomicity property.

**Definition 7 (Service Isolation).** Given services $s_1$, $s_2$, \ldots, $s_n$ satisfying $\varphi_{as}(s_1) = true$, $\varphi_{as}(s_2) = true$, \ldots, $\varphi_{as}(s_n) = true$, $s_1$, $s_2$, \ldots, $s_n$ are called isolated if the atomicity property of $s_1$, $s_2$, \ldots, $s_n$ is still kept when they are running concurrently with shared resources.

In order to describe the implicit interactions and their impacts, we introduce a process $p_s$ for each service $p$ to describe its resource consumption policy and data integrity requirement for shared resources. We call the process $p_s$ as the resource process of service $p$. Suppose a service $p$ uses shared resources $r_1$, $r_2$, \ldots, $r_m$, then the state of its resource process is defined as $\{(x_1, t_1, v_{1}), \ldots, (x_m, t_m, v_{rm})\}$, where variable $x_{ri}$ represents shared resource $r_i$. For any two services $p$ and $q$, we say that $p$ and $q$ can interact implicitly, if $\exists x_{ri}$, such that $x_{ri}$ is a variable in both the states of $p_s$ and $q_s$. On the other hand, for every action $a$ in service $p$, we can define a function $f_i(a)$ to represent the corresponding action in $p_s$ that consumes the shared resources when committing action $a$. If $a$ does not access any shared resource, then $f_i(a)$ is mapped to a silent action $\tau$. We can also specify the data integrity requirement of service $p$ for shared resources in its resource process $p_s$ based on first-order logic. For example, the resource process for service $s_1$ in Fig. 3 can be represented as $p_s = b_2 \cdot (g_1 \cdot e_2) \cdot \vee + b_4 \cdot ((g_1 \cdot e_2) \cdot \vee + \sqrt{v})$, where $g_1$ represents there is no enough resource, and as a result, an exception $e_1$ or $e_2$ is raised when committing the action “$b_2$::query stock” or “$b_4$::produce”, respectively.

By representing a service’s resource consumption policy and data integrity requirement as its resource process, implicit interactions among services can be detected if the states of their resource processes share certain resource variable. Moreover, the interference between two services can be analyzed if one service’s resource process updates the shared resources used by another. To do so, we introduce an operator to compose a service with its resource process as follows:

**Definition 8 (Configuration).** Given a service $p$ and its resource process $p_s$, process $p \oplus p_s$ represents that service $p$ runs under the configuration of $p_s$ for shared resources. The operational semantics of $\oplus$ is given below:

1. If $(p, g_1, a, p') \in F_p$ and $f_{ri}(a) = \tau$, then $(p \oplus p_s, g_1, a, p' \oplus p_s) \in F_{p \oplus p_s}$.
2. If $(p, g_1, a, p') \in F_p$ and $(p_s, g_2, f_{ri}(a), p_s') \in F_{p_s}$, then $(p \oplus p_s, g_1 \land g_2, a, p' \oplus p_s') \in F_{p \oplus p_s}$.
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2876254.

IEEE TRANSACTIONS ON SERVICES COMPUTING, VOL. X, NO. X, JANUARY XXXX

7

requirement. Rule 1 means that the service commits
of resource consumption policy and data integrity
Rule 2 represents that the service commits an action
f

a

Therefore, we compose together a service with its
resource process is detached and released (Rule 4).

Algorithm 1: Locate afflicted implicit interactions.

Input: A service $p$ and its resource process $p_r$, the
public views of collaborating processes $q_{v_1}, \ldots, q_{v_{n-1}}$.

Output: A set of afflicted implicit interactions $S_{AII}$.

1 begin
2 $q_{\text{search}} \leftarrow \{(p, p_r, q_{v_1}, \ldots, q_{v_{n-1}}), \emptyset\}$, $S_{AII} \leftarrow \emptyset$;
3 while $\exists cs \in q_{\text{search}}$ do
4 $q_{\text{search}} \leftarrow q_{\text{search}} - \{cs\}$;
5 $S_{next} \leftarrow \text{getAllNextStates}(cs)$;
6 for $\forall ns \equiv (\{s_0, s_1, \ldots, s_{n+1}\}, t) \in S_{next}$ do
7 $S_{afflicted}, S_{violations} \leftarrow \text{locateAffected}(ns)$;
8 $S_{AII} \leftarrow S_{AII} + \{S_{afflicted}, S_{violations}\}$;
9 if $\text{visited}(ns)$ then
10 $q_{\text{search}} \leftarrow q_{\text{search}} + \{ns\}$;
11 return $S_{AII}$

3) if $(p_s, g, e, p_s') \in F_{p_s} \land e \in E$, then $(p \oplus p_s, g, e, p \oplus p_s') \in F_{p \oplus p_s}$;
4) if $p \in \{0, \sqrt{\cdot}\}$, then $(p \oplus p_s, \top, \tau_s, p) \in F_{p \oplus p_s}$.

The intuitive meaning of Definition 8 is to define
the behavior of a service under the configuration ($p_s$)
of resource consumption policy and data integrity
requirement. Rule 1 means that the service commits
an action which does not access any shared resource;
Rule 2 represents that the service commits an action $a$
and consumes corresponding shared resources (via
$f_r(a)$ from resource process $p_r$); Rule 3 says that if
the data integrity of shared resources for the service
is violated due to implicit interactions (e.g., another
service updates the shared resources to make the
condition $g$ satisfied), an exception is raised and propaga-
ted to the service; if a service is terminated, its
resource process is detached and released (Rule 4).

3.4 Locating Afflicted Implicit Interaction

According to our methodology, each involved service
provider in a service composition needs to analyze
the implicit interactions associated with its service and
the impacts on the service composition. To do so, one
way is to verify all the implicit interactions between a
service composition and other services. However, it is
hard to predict at design time all the possible implicit
interactions that may occur at runtime. We observe
that instead of finding and verifying all the implicit
interactions explicitly at design time, we may check
those places in services where implicit interactions
may occur (that is, hot spots where exceptions will
be raised due to implicit interactions) and evaluate
their impacts on the atomicity property of services.
Therefore, we compose together a service with its
resource process to verify whether the resulting com-
posite process satisfies the atomicity sphere. If not,
then we can find out under what conditions and
which implicit interactions cause atomicity violations.

According to Definition 5, checking the atomicity
sphere of a service composition needs a traversal of
its every trace to verify whether there exists a non-
compensable task that need to be compensated due
to exception handling. Algorithm 1 consists of two
parts: Part 1 (Algorithm 2) illustrates how to construct
the traces of a service composition based on symbolic
execution [13]; Part 2 (Algorithm 3) identifies the tasks
that need to be compensated caused by an exception,
and marks those non-compensable ones as afflicted.

Given the current state, Algorithm 2 generates all
the feasible next states based on symbolic execution of
the service composition. Lines 3 to 6 represents that
$p$ commits a task and transits to its next state if its
resource process satisfies the conditions. Lines 7 to 10
represents that the collaborating service $q_i$ commits a
task and transits to its next state.

Given a service instance, Algorithm 3 checks
whether the current instance violates the atomicity
sphere due to the exceptions raised from the resource
process (represented as $E_{rs}$ in the algorithm). Lines 4
to 8 and Lines 9 to 13 check the two conditions
in Definition 5, respectively. That is, if $t$ violates
the atomicity sphere due to some exception $e_i$, then
the potential implicit interactions associated with the
hot spot where $e$ is raised are marked as afflicted.
The non-compensable actions that lead to atomicity

Algorithm 2: getAllNextStates($cs$)

Input: A symbolic state of a service composition
$p \oplus p_s \parallel q_{v_1} \parallel \cdots \parallel q_{v_{n-1}}$, that is,
$cs \equiv (\{s_0, s_1, \ldots, s_n\}, t)$, where
$s_0, s_1, s_i(2 \leq i \leq n)$ are the current state of
$p, p_s$, and $q_{v_{i-1}}$, respectively, and $i$ is the
current trace of the instance;

Output: A set of next available states after
committing one action $S_{next}$;

1 begin
2 $S_{next} \leftarrow \emptyset$;
3 for $\forall (s_0, g_0, a_0, s_0') \in F_{s_0}$ do
4 if $\exists (s_1, g_1, f_r(a_0), s_1') \in F_{s_1} \land t \models (g_0 \land g_1)$
5 then $t' \leftarrow t_0f_r(a_0)$;
6 $S_{next} \leftarrow S_{next} + \{\{s_0', s_1', \ldots, s_n\}, t'\}$;
7 for $\forall (s_i(2 \leq i \leq n))$ do
8 if $\exists t \models a_i \rightarrow s_i'$ then
9 $t' \leftarrow t_0$;
10 $S_{next} \leftarrow S_{next} + \{\{s_0, s_1, \ldots, s_i', \ldots, s_n\}, t'\}$;
11 return $S_{next}$;
Algorithm 3: locateAffected(cs)

Input: A symbolic state of a service composition
\[ p \parallel p_s \parallel q_{v_1} \parallel \cdots \parallel q_{v_{n-1}}, \] that is,\n\[ cs = (s_0, s_1, \ldots, s_n, t), \]
where
\[ s_0, s_1, s_t (2 \leq t \leq n) \] are the current state of \( p, p_s, \) and \( q_{v_{i-1}}, \) respectively, and \( t \) is the current trace of the instance;

Output: A set of afflicted implicit interactions \( S_{\text{affected}} \) and actions \( S_{\text{violations}} \) that violate atomicity;

\[
\begin{array}{l}
\text{begin} \\
S_{\text{affected}} \leftarrow \emptyset; \\
t_{a_m} \leftarrow t; \\
\text{if } \exists e \in E_{rs} : a_m \equiv \tau_e \text{ then} \\
\quad \text{for } \forall t'[i] \text{ do} \\
\quad \quad \text{if } \text{reach}(t'[i], e) \land NC(t'[i]) \text{ then} \\
\quad \quad \quad S_{\text{affected}} \leftarrow S_{\text{affected}} \cup \{e\}; \\
\quad \quad \quad S_{\text{violations}} \leftarrow S_{\text{violations}} \cup \{t'[i]\}; \\
\quad \text{if } \exists e \in E_{rs} : a_m \equiv e \text{ then} \\
\quad \text{for } \forall t'[i] \text{ do} \\
\quad \quad \text{if } NC(t'[i]) \text{ then} \\
\quad \quad \quad S_{\text{affected}} \leftarrow S_{\text{affected}} \cup \{e\}; \\
\quad \quad \quad S_{\text{violations}} \leftarrow S_{\text{violations}} \cup \{t'[i]\}; \\
\text{return } S_{\text{affected}}, S_{\text{violations}};
\end{array}
\]

violations are also recorded \((S_{\text{violations}})\).

Note that in Algorithm 2, the operational semantics introduced in Section 2.2 are applied to derive the transitions of a service (e.g., Line 3). As illustrated in Section 2.3, the reach relation between two actions can be derived when we apply the operational semantics to derive a service’s transitions. Such information can be cached in some hash table for later usage (i.e., Algorithm 3, Line 6). Algorithm 1 needs to traverse every state of a service composition. In our work, we assume that a service composition should terminate [24]. Let \( m \) be the number of states of a service composition, the complexity of Algorithm 1 is \( O(m) \).

3.5 Resolution of Atomicity Violation

Based on the \( S_{\text{affected}} \) and \( S_{\text{violations}} \) reported by Algorithm 1, service consumers and service providers obtain the atomicity violation scenarios that could occur in the concerned service composition due to implicit interactions. Preventive measures can be then designed to avoid such scenarios or resolve the atomicity violations. However, from the perspective of service providers, avoiding such implicit interactions may compromise their capability to provide services (since this would reduce the concurrent execution rate of services). A promise mechanism proposed by Greenfield et al. could help to solve this problem [8], [11]. A service consumer may request a promise from a service provider to agree on some conditions (some afflicted implicit interactions will not occur) when providing its service to the service consumer. In return, the service provider may charge the service consumer based on the promise it requests (e.g., a higher fee for suppressing more afflicted implicit interactions). In Sifter, two options are provided to implement the promise of avoiding implicit interactions, namely exception handling and behavior constraints.

3.5.1 Exception Handling

According to Definition 5, an exception may cause a service violate the atomicity sphere if the exception is not captured or not handled properly so that some non-compensable actions need to be compensated. Therefore, a straightforward solution is to add extra exception handlers to handle the exceptions raised by implicit interactions. From the syntactic point of view, one can always add some exception handlers to handle the exceptions raised by implicit interactions and make a service composition satisfy the atomicity sphere. That is, once service providers configure their services by enacting new exception handlers to handle the exceptions raised by afflicted implicit interactions, they can apply Algorithm 1 to check whether the new enacted exception handlers satisfy the atomicity sphere. This can be done interactively with locating afflicted interactions and enacting new exception handlers until the atomicity sphere is satisfied.

However, an exception handler also needs to concern about the functionality and semantics of a service composition, which may not always be available in practice. For example, if we design an exception handler to handle the exception raised in Fig. 3 by re-ordering the missing resources, the service composition will not violate the atomicity requirement. However, such an exception handler may not be feasible due to the application requirements (e.g., it may take a long time to re-order the resources so that the supplier fails to fulfill the contract with the retailer within the required time). Therefore, a complementary solution based on behavior constraints is needed.

3.5.2 Behavior Constraint

Another approach to suppressing the influence of afflicted implicit interactions is to impose some execution restrictions to services so that those afflicted implicit interactions will be removed or their impacts on the atomicity property of a service composition will be suppressed. To do so, we propose to specify behavior constraints for services to obey:

Definition 9 (Behavior constraint). A behavior constraint is a first order logic formula over a set of service composition instances in the following format: \( \forall t_1, \ldots, t_n, f(t_1, t_2, \ldots, t_n) = \text{true} \), where \( t_i \) is an instance of some service composition \( s_i \), and \( f \) is a predicate.
A behavior constraint quantifies a safety property amongst services when they are running concurrently. Whenever a task is to be executed, it will be checked first if its execution will violate any behavior constraints. The execution of a task will not start until its execution will not breach any behavior constraint.

More specifically, let $d$ be a data integrity requirement of service $s$ that needs to be held between tasks $a$ and $b$ ($b$ can be equal to $a$). For any task $c$ of $s'$, if the execution of $c$ causes $d$ to be violated, the behavior constraint is to forbid $c$ from being executed between $a$ and $b$. For example, one could design a behavior constraint in Fig. 1 to forbid the commitment of task “$b_1$:Read BOM” until the design service terminates if task “$a_2$:Write BOM” has been committed.

The duration that a behavior constraint needs to be held for a service is critical to its performance (e.g., concurrent execution rate). To shorten the duration for a behavior constraint and improve the performance of services, we propose to relax behavior constraints using the information of atomicity violations caused by implicit interactions. Let us revisit the example in Fig. 1, where the atomicity property may be violated due to the implicit interaction between the design service $s_1$ and the manufacturing service $s_2$, because task “$a_3$:Test” may cause an exception for task “$b_1$:Read BOM”, and the handling of the exception requires compensating task “$c_2$:Produce items”, which however is non-compensable. Therefore, we can specify a relaxed behavior constraint as follows to avoid the atomicity violation: task “$c_2$:Produce items” should not be committed unless task “$a_3$:Test” has been committed. This behavior constraint is more relaxed than the previous one because some tasks (e.g., “$b_1$:Read BOM”, “$b_2$:Assign tasks” and “$c_1$:Receive tasks”) need not wait the termination of the design service.

Enforcing above relaxed behavior constraint however is difficult, because tasks “$a_3$:Test” and “$c_2$:Produce items” are from different services located in autonomous and distributed organizations. To alleviate this problem, we propose to propagate the compensability property of the tasks in $S_{violations}$ to every service involved in a service composition, if these tasks need to be compensated due to the implicit interactions associated with the service. The purpose is to use the propagated property to enforce local behavior constraints without involving tasks from distributed services. For example, as depicted in Fig. 1, if we propagate the non-compensable property of task “$c_2$:Produce items” to task “$b_2$:Assign tasks” (such that “$b_2$:Assign tasks” becomes non-compensable) in the manufacturing service, then we can use task “$b_2$:Assign tasks” to substitute the role of “$c_2$:Produce items” in the above behavior constraint. In this way, “$c_2$:Produce items” will not be committed unless “$a_3$:Test” has been committed.

Suppose an afflicted implicit interaction associated with $s_i$ causes atomicity violations in the service composition $s$ (of $s_1, \ldots, s_n$), and $S_{violations}$ is the set of non-compensable tasks that need to be compensated due to this implicit interaction. Let $A_i$, $H_i$ be the set of actions and port actions of service $s_i$, respectively. For any task $a \in S_{violations} \land a \notin A_i$, the rule for propagating the compensability property of $a$ to service $s_i$, denoted as $s'_i = \mathbb{R}(a, s_i)$, is given as follows. To ease the presentation, given any trace $t$ of $s$, we denote the temporal order of two tasks as $t[i] \leq t[j]$, if and only if $1)$ $t[i], t[j] \in A_k \land i < j$, or $2)$ $\exists t[i]: t[i] \leq t[l] \land t[l] \leq t[j]$.

Definition 10 (Property propagation rules). Two rules are defined to propagate the compensability property for behavior constraints:

Rule 1: $\forall t \in trace(s), if a = t[g] \land (\exists t[h] \in H_i : t[h] \leq t[l] \land t[l] \leq t[g]), mark t[h]$ as non-compensable.

Rule 2: $\forall t \in trace(s), if a = t[g] \land (\exists t[l] \in H_i : t[l] \leq t[g]) \land t[k] \in A_k (0 < h < g) \land (\forall k (0 < k < h) : t[k] \notin A_k), mark t[h]$ as non-compensable.

Intuitively, Rule 1 means that the non-compensable property of $a$ is propagated to $s_i$ along the explicit interactions among the services recursively. For example, the non-compensable property of task “$c_2$:Produce items” in Fig. 1 is propagated to “$b_2$:Assign tasks” via the explicit interaction between the manufacturing service and the workshop service. If there is no explicit interaction before $a$, the property of $a$ is propagated to the initial task of $s_i$ (Rule 2). Note that these two rules can be applied to Algorithm 1 during the atomicity analysis of a service composition.

Based on above discussions, we can refine behavior constraints in the following way, as illustrated in Algorithm 4. Suppose an afflicted implicit interaction associated with $s_i$ violates the data integrity requirement $d$ of $s_i$ and causes atomicity violations in the service composition $s$ (of $s_1, \ldots, s_n$). Let $s'_i$ be the resulting service of $s_i$ after property propagation, and
be the exception raised by task \( b \) in \( s_i \) due to the violation of \( d \). For any task \( a \) of \( s' \), satisfying \( reach(a, e) \land NC(a) \), if task \( c \) from a different service (or service instance) violates \( d \), \( c \) is not allowed to commit between the period between \( a \) and \( b \).

### 3.6 Practical Concerns

The general idea and algorithms of Sifter for service-based Internet applications presented in previous sections can be implemented in different service frameworks. In this section, we take BPEL as an example to discuss its implementation issues.

The exception handling mechanism is well supported in BPEL based on the built-in “catch” primitive and user-defined fault handlers. One major practical concern is about the efforts needed to deal with the afflicted implicit interactions using exception handling, including how to organize the tasks in the fault handlers to ensure the atomicity of applications, and how to design fault handlers consistent with the semantics of applications. Designers can apply Algorithm 1 recursively to verify whether the fault handlers satisfy the atomicity requirement. The semantics of fault handlers however is related to application requirements and usually needs more human efforts.

To enforce the behavior constraints for BPEL, we implement a BPEL engine plug-in for Apache Axis2 server. That is, by inheriting the BPEL runtime implementation, the plug-in overloads the activity scheduler to check the behavior constraints before the execution of an activity is enabled. Since the plug-in is transparent to BPEL process designers, they need no extra efforts to enforce the behavior constraints. To implement the BPEL engine plug-in, we employ an event-driven solution to enact and release the behavior constraints at runtime. That is, as illustrated in Fig. 5, on receiving a scheduling request from the engine, the scheduler issues an event to invoke the BC checker to verify the monitored behavior constraints. If no behavior constraints are violated, the scheduler issues another event to invoke the BC dispatcher to attach available behavior constraints to the monitored queue before the activity is scheduled. On receiving the completion event of an activity, the BC dispatcher is invoked again to release the monitored behavior constraints that are not available any more.

### 4 Evaluation

This section evaluates the performance, effectiveness and scalability of our proposal.

#### 4.1 Performance Comparison

To investigate the performance overhead and the efficiency of resource usage, we make the overall comparison between our approach and three baseline approaches with two sets of experiments. In particular, Approach 1 (denoted as sequence) avoids implicit interactions by executing services in a sequential manner; Approach 2 (denoted as serialize) imposes behavior constraints for all the implicit interactions, representing a serializable scheduling of services; Approach 3 (denoted as sandbox) applies sandboxes to separate resource among services. These three baseline approaches represent the two extreme solutions that either avoid concurrent execution of services or forbid resource sharing among services. Differently, Sifter imposes behavior constraints to suppress afflicted implicit interactions only. In the experiments, we randomly generated a pool of 10,000 services modeled and adapted from business applications in the widely used TPC-C benchmark. In particular, each service has 50 tasks and the time duration for each task was randomly set to a value between 1 and 10 time units (we set 1 second as a time unit in the experiment).

In the first experiment, we randomly selected 100 services from the service pool and randomly added one resource-conflicting implicit interaction to each service. Then, we changed the number of services whose implicit interactions are afflicted and studied the impacts on the performance overhead. To do so, we randomly set part of the services (with a pace of 10 percentage of the total services) whose implicit interactions are afflicted and applied the aforementioned approaches to schedule the 100 services. For each configuration, the experiment was repeated 128 times and the average results are used to measure the performance overhead. To measure the performance overhead, the experiment recorded the total time needed for executing a given number of services, the average response time and the average completion time for these services. To investigate how the efficiency of resource usage may affect the performance, we ran the sandbox solution twice for each experiment configuration, assigning each sandbox 10 percentage of the total resources (minimal for executing a service, denoted as sandbox(10%)) and 20 percentage of the total resources (denoted as sandbox(20%)), respectively.

4. TPC benchmark, available at: http://www.tpc.org/. The adaptation includes the number of tasks for a business process, the time period for each task, the evaluation metric of performance, and the business scenarios.
In the experiment, the total time for executing all the 100 services took about 7 hours using the sequence solution, and its average response time and completion time were nearly 4 hours. The results of the other approaches are shown in Fig. 6(a-c). We can observe that Sifter has the smallest average completion time and total time than others, especially when the percentage of services with afflicted implicit interactions is small. We can also observe that both the average completion time and total time of the serialize solution are smaller than that of sandbox(10%). This indicates that the sandbox solution uses the resource less efficiently than the serialize solution does, since the resource cannot be shared among different services once it is assigned to a sandbox. Moreover, the more resource is assigned to a sandbox, the less efficiently the resource can be used. As a result, sandbox(20%) has a worst performance than sandbox(10%). Differently, Sifter suppresses the afflicted implicit interactions only and thus uses resource more efficiently to gain the best performance in terms of average completion time and total time. In addition, Fig. 6(a) shows that in most cases, Sifter has the smallest average response time, but the difference among Sifter, serialize and sandbox(10%) is not significant. This is because the average response time is determined by the first task of each service in the experiment setting, but the first task is less likely to have implicit interactions. As a result, the benefit of Sifter is not obvious. However, due to the low efficiency of resource usage, the average response time of sandbox(20%) is larger significantly than that of all the other solutions.

In the second experiment, we randomly selected 100 services from the service pool and randomly assigned 10 resource-conflicting implicit interactions to each service. Then, we changed the number of afflicted implicit interactions a service owns, and studied its impacts on the performance overhead. We started the experiment by assuming that none of them are afflicted, and repeated the experiment by increasing the number of afflicted implicit interactions with a pace of one (i.e., 10 percentage of the total implicit interactions a service owns) each time. We applied the aforementioned approaches to schedule the 100 services and each configuration was repeated 128 times. The average results are used to measure the performance overhead.

The results of the second experiment are depicted in Fig. 6(d-f). We can observe that Sifter has a better performance than all the other solutions in terms of the total time and the average completion time: the less percentage of afflicted implicit interactions, the better performance benefits Sifter can achieve. We can also observe that the total time of sandbox(10%) is larger than that of the serialize solution, but its average completion time is smaller than that of the serialize solution. This is because a service scheduled by the serialize solution may have to wait for other services to release the resource during their scheduling. Differently, a service scheduled by the sandbox solution needs not wait for other services to release the resource once a sandbox is assigned to it. As a result, its average completion time is smaller than that of the serialize solution, although the serialize
solution has a better resource usage efficiency than the sandbox solution does. However, such benefits are overwhelmed by the efficiency loss of resource usage in sandbox(20%). Therefore, the serialize solution has a smaller average completion time than that of sandbox(20%). In addition, Fig. 6(d) shows that Sifter has a smaller average response time than that of the sandbox solutions, but the difference of the average response time between Sifter and the serialize solution is not significant. This is because the average response time is determined by the first task of each service only in the experiment setting, but the first task is less likely to have an afflicted implicit interaction. Differently, before scheduling the first task of a service, the sandbox solution may need to request for available resources to create a sandbox for the service. This may take a long time especially when the efficiency of resource usage is low. Therefore, both sandbox solutions have a larger average response time than that of Sifter.

4.2 Runtime Complexity

To evaluate the overhead of checking behavior constraints, we designed a service with three behavior constraints for resource-conflict implicit interactions. Then, we scheduled the service with the number of service instances varying from 10 to 200. We recorded the number of constraint checking (BC#) in each run of the experiment and the overall time spent in checking all these constraints, as shown in Table II. The results indicate that the overhead is quite small, and can be ignored in practice (e.g., 0.46 million checking of behavior constraints only took less than 1 second).

To evaluate the scalability of Algorithm 1, we designed an experiment to investigate and evaluate the factors on the time spent in locating afflicted implicit interactions. First, we generated a set of traces with the number of tasks varying from 10,000 to 100,000 with a pace of 10,000. We assumed that the service composition involves two services. For every task in one trace, the possibility of being a port action is 10%. In each trace, we randomly selected one task that forms an implicit interaction with other services and recorded the time needed to locate implicit interactions for each trace using Algorithm 1. Next, we studied the impact of another factor by varying the number of traces in a service composition. To do so, we fixed the number of tasks in each trace (e.g., 500), increased the number of traces from 10,000 to 1,000,000 with a pace of 10,000 and recorded the time needed to locate the afflicted implicit interactions using Algorithm 1. Finally, we studied whether the number of services involved in a service composition affects the time spent in locating afflicted implicit interactions in a trace. We fixed the number of tasks in a trace (e.g., 10000), varied the number of services involved in the service composition from 2 to 20 with a pace of 2, and recorded the time needed to locate the implicit interactions in the trace using Algorithm 1.

We can observe from Fig. 7(a) that the time spent in locating afflicted implicit interactions in one trace is nearly a quadratic function of the number of tasks in the trace. Fig. 7(b) shows that the time is almost linear to the number of traces when the number of tasks is fixed in each trace. Fig. 7(c) further confirms that the more services get involved in an application, the more time is needed to locate afflicted implicit interactions (since the more states needed to be traversed).

5 RELATED WORK

In cloud environments, Internet applications are usually composed of distributed, self-contained and autonomous services, imposing great challenges for the development, deployment and maintenance of Internet applications [28], [29]. Methodologies such as DevOps [16] are proposed to manage the complexity of such applications. One important principle of DevOps is to apply containerization tools such as Docker [17] to package the resource required for applications so that applications can be quickly deployed and migrated to the changing environments. Note that Sifter can also be regarded as a container, in the sense that the behavior constraints packaged with the services are used to hold the required resource for services. Compared to Docker, Sifter is a more light-weighted and fine-grained container because only those shared resource that may affect the application consistency is packaged to isolate their interference. As a result, Sifter has a better performance and efficiency of resource usage, as indicated in the experiment.

Recently, the isolation issue in cloud computing has also attracted many researchers’ attentions from different perspectives such as storage [7], [26], privacy and security [4], [27], virtual machines [14], [23], data center and network [18], [25], and so on. Compared to the isolation at the infrastructure level in the aforementioned work, Sifter focuses on the isolation at the service level and is thus more fine-grained. In addition, Loesch et al. proposed a solution to isolate performance in multi-tenant systems based on the request admission control on identified positions in the architecture [15], but their solution does not concern the consistency of applications. Differently, Sifter isolates services to work correctly to preserve their application consistency and concerns both the performance and the efficiency of resource usage.

Various techniques for detecting and preventing data races in concurrent systems and cloud computing

<table>
<thead>
<tr>
<th>BC#</th>
<th>Time (ms)</th>
<th>BC#</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9369</td>
<td>&lt; 1</td>
<td>187,100</td>
<td>219</td>
</tr>
<tr>
<td>25,129</td>
<td>15</td>
<td>207,847</td>
<td>281</td>
</tr>
<tr>
<td>53,351</td>
<td>16</td>
<td>274,693</td>
<td>534</td>
</tr>
<tr>
<td>118,740</td>
<td>47</td>
<td>341,895</td>
<td>394</td>
</tr>
<tr>
<td>135,592</td>
<td>110</td>
<td>462,241</td>
<td>985</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BC#</th>
<th>Time (ms)</th>
<th>BC#</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>135,592</td>
<td>47</td>
<td>462,241</td>
</tr>
<tr>
<td>42</td>
<td>1939-1374</td>
<td>42</td>
<td>219</td>
</tr>
<tr>
<td>17</td>
<td>864</td>
<td>17</td>
<td>281</td>
</tr>
<tr>
<td>31</td>
<td>607</td>
<td>31</td>
<td>534</td>
</tr>
<tr>
<td>110</td>
<td>15</td>
<td>110</td>
<td>394</td>
</tr>
<tr>
<td>985</td>
<td>219</td>
<td>985</td>
<td>985</td>
</tr>
</tbody>
</table>

have also been proposed (e.g., [5], [12], [33], [34]). Note that the resource conflict may be considered as a kind of data race. However, it is difficult to apply these techniques to eliminate the impacts of resource conflicts among different autonomous, heterogeneous, loosely coupled and generally long running services that are across different organizations. For example, Xu et al. [34] proposed a static analysis approach to detect data race in BPEL web services based on constraint solver. By constructing a happen-before graph, constraints such as READ-WRITE are derived and solved using existing constraint solver to find the possible data race scenarios. Sifter also uses a static approach to analyze the potential implicit interactions. The difference is that their work focuses on detecting data races inside a BPEL service, whereas Sifter is not limited to find out the implicit interactions inside a single service. In addition, Sifter also propagates the compensability property from one service to another to enact behavior constraints for service compositions across different organizations, which is not covered in existing data race detection approaches. Jia et al. [12] also proposed a dynamic analysis-as-a-service using a multi-virtual machine architecture, namely sda-cloud, to detect dynamic data race in the cloud. In sda-cloud, each VM hosts a different compiled version of the same program component and runs a state-of-the-art detector to detect data races. This is similar to the Docker solution but its main purpose is to analyze the executions of program components under different configurations using multiple VMs effectively with a set of VM selection strategies. The Sda-cloud architecture may be used to configure the scenarios where services are composed from different organizations, but it is mainly used to detect all the possible data races under different configurations. Differently, Sifter focuses on locating only those situations that can breach application consistency among different services or instances, and resolving their impacts based on exception handling and behavior constraints using property propagation.

One may also apply transactional protocols such as BTP [20], WS-TX [21] to safeguard the application consistency of collaborative applications. These protocols however did not address how to detect the potential atomicty violation scenarios caused by implicit interactions. Some extensions to WS-TX protocol including [1], [10] addressed the implicit interactions based on a global transaction dependency graph, which however did not consider the atomicty violations and how to customize the quality of services. In addition, research efforts were made to study the isolation issue in transactional workflow based on the serializability of processes [24]. This however can lead to poor performance and is therefore unacceptable for long-running services [6], [32]. Some studies were also conducted to relax the serializability criterion using application semantics [2], [22]. These approaches could be applied to handle the resource-conflict implicit interactions. The difference is that they addressed the issue from the perspective of completing a transaction. If a resource conflict does not change the outputs of applications at the end, such an interaction, according to these studies, does not have to be handled. However, such resource conflict may still lead to atomicty violations in a service composition. We study these implicit interactions from the perspective of aborting a transaction instead of completing it. Obviously, both perspectives should be addressed. Our approach thus complements theirs.

6 CONCLUSION

Service isolation is desirable to avoid interference among Internet applications but it usually compromises either the service performance or the efficiency of resource usage. This paper presents Sifter, a novel fine-grained strategy to isolate services for Internet applications concerning the balance of service performance and resource usability. We have formulated a process algebra framework to integrate services’ behavior with their resource processes to study their implicit interactions. Algorithms are designed to locate the afflicted implicit interactions that may breach the application consistency of Internet applications. By introducing exception handlers or behavior constraints to the afflicted implicit interactions, Sifter eliminates the impacts of those afflicted implicit interactions only and filter the others. As a result, services are isolated to work correctly with an acceptable overhead. The experimental results show that Sifter outperforms baseline solutions in terms of a lower performance.
overhead and a higher efficiency of resource usage. Currently, Sifter can only be applied to applications whose back-end processes are statically defined. In the future work, we plan to extend Sifter to support self-adaptive and artifact-centric services in dynamic service compositions.

REFERENCES


Chungyang Ye received the PhD degree from the Hong Kong University of Science and Technology in 2008. He is now a professor in the College of Information Science and Technology, Hainan University. His research interests include software engineering issues on business process management, distributed and pervasive middleware systems, services and cloud computing.

S.C. Cheung received the MSc and PhD degrees in computing from Imperial College London. He is a professor in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology. His research interests include context-aware computing, service-oriented computing, software testing, fault localization, RFID, and wireless sensor network systems.

S.C. Cheung received the MSc and PhD degrees in computing from Imperial College London. He is a professor in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology. His research interests include context-aware computing, service-oriented computing, software testing, fault localization, RFID, and wireless sensor network systems.

W.K. Chan received the PhD degree in software engineering from the University of Hong Kong in 2004. He is an associate professor in the Department of Computer Science at City University of Hong Kong. His current research interests include issues in software testing and analysis, service engineering, and development of context-aware applications and wireless sensor network applications.