Detection and Resolution of Atomicity Violation in Service Composition

Chunyang Ye and S.C. Cheung*
Dept. of Comp. Sci. & Eng.
Hong Kong Univ. of Sci. & Tech.
Hong Kong, China
{cyye, scc}@cse.ust.hk

W.K. Chan
Dept. of Comp. Sci.
City Univ. of Hong Kong
Hong Kong, China
wkchan@cs.cityu.edu.hk

Chang Xu
Dept. of Comp. Sci. & Eng.
Hong Kong Univ. of Sci. & Tech.
Hong Kong, China
changxu@cse.ust.hk

ABSTRACT
Atomicity is a desirable property that safeguards application consistency for service compositions. A service composition exhibiting this property could either complete or cancel itself without any side effects. It is possible to achieve this property for a service composition by selecting suitable web services to form an atomicity sphere. However, this property might still be breached at runtime due to the interference between various service compositions caused by implicit interactions. Existing approaches to addressing this problem by restricting concurrent execution of services to avoid all implicit interactions however compromise the performance of service compositions due to the long running nature of web services. In this paper, we propose a novel static approach to analyzing the implicit interactions a web service may interfere and their impacts on the atomicity property in each of its service compositions. By locating afflicted implicit interactions in a service composition, behavior constraints based on property propagation are formulated as local safety properties, which can then be enforced by the affected web services at runtime to suppress the impacts of the afflicted implicit interactions. We show that the satisfaction of these safety properties exempts the atomicity property of this service composition from being interfered by other services at runtime. The approach is illustrated using two service applications.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability.*

General Terms
Algorithms, Reliability, Theory, Verification.

Keywords
Atomicity, behavior constraint, implicit interaction, web services.

1. INTRODUCTION
Service oriented architecture (SOA) is an emerging software engineering paradigm for developing distributed applications in the Internet era. In this paradigm, individual service providers develop their web services, and publish them at service registries. Service consumers could then discover their required web services from the service registries and compose them to implement their own web services, or utilize them to support their business activities. Usually, each web service is driven by a backend process that describes its underlying workflows.

As web services are distributed, loosely coupled, heterogeneous and autonomous, an important concern in service compositions is application consistency [6]. For instance, in a supply chain application, a retailer composes services from suppliers and a bank. Suppose the retailer has paid an order but the supplier fails to deliver the cargo to the retailer. Then application inconsistency may arise if the supplier service does not refund the retailer. To avoid such kinds of undesirable application inconsistency, *atomicity property* upon a service composition should be in place. This means that the composite service could either complete the expected business transaction, or cancel it without any side effect by compensating all committed tasks through service recovery. A service composition could achieve this property by making itself an *atomicity sphere* [7], a structural criterion and sufficient condition for atomicity of workflow and web services.

To check whether a service composition forms an atomicity sphere without disclosing certain selected details of involved services, our previous work [16] proposed an approach to constructing public views from their backend processes. Such a public view encapsulates tasks undesirable to expose to the public and remain atomicity-equivalent to its backend process. Service consumers can use these public views to check whether an intended service composition form an atomicity sphere. In [17], we further proposed a decentralized approach to checking this property.

These approaches [16][17] assume no interference between service compositions. However, this assumption may not always hold. Services can interact in *explicit* and *implicit* manners. Explicit interactions occur when services communicate through their communicating ports. This is the way that multiple services are aggregated to form a service composition. Implicit interactions occur when services exchange information via shared resources, such as a common data repository. For instance, a product design service may share with a production service a design document stored in the back office of an enterprise. Such implicit interactions are common to services provided by the same organization. However, they may cause interference between service compositions at runtime and lead to atomicity violation, even if each of these service compositions forms an atomicity sphere.

ESEC-FSE'07, September 3-7, 2007, Cavat near Dubrovnik, Croatia.

* Correspondence author

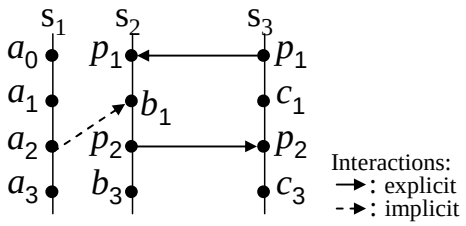

Figure 1. Impacts of implicit interaction on atomicity.

Let us illustrate the problem using the example in Figure 1, where there are a total of three web services, s_1, s_2, and s_3. Services s_1 and s_2 are provided by a subprime lending company. In particular, s_1 is a corporate finance service which borrows large loans from the other banks, and s_2 is a retail lending service for providing individual loan to customers. There is one implicit interaction between tasks a_2 of s_1 and b_1 of s_2 (represented as a dashed arrow), because a_2 determines (negotiates with the other banks) how much money (credits) it may borrow from other banks, and b_1 assesses the credit risk to decide how much money it may lend to a particular customer. Service s_3 is a customer agency service which composes s_2 to borrow money from the subprime lending company. Suppose that the composition of service s_2 and s_3 satisfies atomicity sphere to guarantee any individual execution of this service composition to either succeed or fail without any side effects. However, when services s_1 and s_2 are executed concurrently, task b_1 in service s_2 may use the result (i.e., credits) of task a_2 in service s_1. In this scenario, if task a_3 fails (e.g., the banks refuse to lend money to the company because they discovered this company has a lot of bad debt, or because the inter-bank interest rate rises abnormally), then s_1 will rewind to its original state and compensate all its committed tasks. Service s_2 also needs to rewind and re-evaluate the loan request and determine the loan it may approve because the credits change due to the failure of a_3. Therefore, the subsequent tasks of b_1 also need compensating and re-executing. However, the subsequent task c_3 may have already committed in s_3 and its effects are unable to compensate (e.g., uses the approved credit to buy a house in advance and incurs penalty if the transaction is cancelled), then atomicity is violated in this situation in the composition of services s_2 and s_3.

A simple approach to preventing such a scenario is to forbid any concurrent execution of web services. However, this is impractical because it will seriously compromise the performance and interests of service providers. Similarly, it is also impractical to handle these implicit interactions using conventional transaction approaches [12], because web services are generally long running. Moreover, web services are reusable components. Managing such implicit interactions of a service in all its possible compositions in the same way will reduce its reusability (since different service composition has different requirement). An alternative solution is to locate the potential afflicted implicit interactions that would affect the atomicity property in each of its service compositions and eliminate their impacts individually to preserve the atomicity property. However, this approach is non-trivial because it is difficult to predict in advance all the afflicted implicit interactions that may occur at runtime due to the open environment of web services.

The main contributions of this paper are three-folded. First, a novel approach is proposed to statically analyzing the potential atomicity violation scenarios in a service composition caused by implicit interactions. Based on the def-use relationships among tasks, afflicted potential implicit interactions are located by analyzing their potentially affected tasks in a service composition. Second, based on these analysis results, an approach based on behavior constraints is proposed to resolve the atomicity violation scenarios in a service composition. We add local behavior constraints as safety properties to the involved web services to facilitate this process based on property propagation. We prove that satisfying these local safety properties in each involved web service can eliminate the impacts of afflicted implicit interactions and guarantee the global atomicity property in the composition of these services. Third, algorithms are presented to compute and implement the local behavior constraints for web services.

The rest of this paper is organized as follows: Section 2 introduces some preliminary concepts about services and atomicity sphere. Section 3 presents our methodology and its theoretical results. Section 4 evaluates our proposal by two real life examples, analyzes the feasibility and discusses the limitations of our current work. This is followed by a comparison with related works in Section 5, and finally Section 6 concludes this paper.

2. PRELIMINARY

In this section, we introduce a few concepts: process, atomicity-equivalent public view, and atomicity sphere. We denote a web service by its backend process.

Definition 1 [Process]: A process p is defined by a quadruple (p, P, A, F), where P is a set of states, A is a set of actions, $F \subseteq (P \times A \times P)$ is a ternary transition relation. For simplicity, we denote the initial state of a process p as p, and p^* as all the states reachable from p. Without loss of generality, we represent the termination state of a process as the constant 0. We assume that each process should terminate [12].

Processes can be composed by several basic operators, that is, "$\cdot$", "$\|$", and "+", representing the prefix operator, parallel composition operator and choice composition operator, respectively [16]. For example, a process $p = a \cdot q$ represents that after committing the action a, the process p reaches the state q. If p is equal to q, that is, $p = a \cdot p$, then this formula represents a recursive process (that is, a loop). A *service composition* of services s_1, s_2, …, s_n is the parallel composition of the backend processes of these services, that is, $s = s_1 \| s_2 \| \ldots \| s_n$.

In this paper, an *action* refers to the commitment of a task, which occurs instantaneously. To ease the presentation, the terms "*action*" and "*task*" are used interchangeably. In a process, *port actions* are designed to communicate with other processes explicitly, while *non-port actions* do not participate in any explicit communication (that is, explicit interaction). We assume in this paper that processes communicate with each other synchronously at the application level (although the underlying communication protocols may be asynchronous). For example, in a supply chain application, the customer process may send an order to the supplier process using asynchronous protocols (e.g., email), but the customer does not conduct next action until it receives the acknowledgement from the supplier. Hence, the port action "*place order*" from the customer and the port action "*receive order*" from the supplier communicate synchronously in the application level. To ease the presentation, we assume port actions with the same name from two processes communicating with each other.

Definition 2 [Trace]: A trace of a process is a sequence, possibly empty, of actions reachable from its initial state. For a process *p*, *trace*(*p*) denotes the set of all the traces of process *p*.

For a trace *t*, *i* is an integer larger than *0*, *t*[*i*] denotes the action committed in the i^{th} position of *t*. Let *B* be a set of actions, *t**B* represents the trace removing all the actions from *t* except those in *B*. For an action *t*[*i*]∈*B*, function *location*(*t*, *i*, *B*) returns the new position of the action *t*[*i*] in the trace *t**B*, that is, *location*(*t*, *i*, *B*) = *i* – *m*, where *m* is the number of actions *t*[*j*] (0< *j* < *i*, and *t*[*j*]∉*B*). For completeness, if *t*[*i*]∉*B*, then *location*(*t*, *i*, *B*) = *0*. In this paper, we refer to an execution of a service as an instance of this service. To ease the presentation, we represent an *instance* of a service by a trace. Let *trace*(*s*/s_i) denote the set of instances of service s_i that engage in the service composition $s = s_1||s_2||\ldots||s_n$.

To model the atomicity sphere of a process, every action *a*∈*A* is assigned two properties: *compensability* and *retriability*. A *compensable* action *a*, denoted as C(*a*) = *true*, means that this task can be undone after committing. An action is *non-compensable* if the overhead or cost of compensating this task is unacceptable [7]. A *retriable* action *a*, denoted as R(*a*) = *true*, can repeat itself internally without cumulative effects if the latest internal trial fails, and finally succeed after a finite number of trials; otherwise, the action is *non-retriable*. Informally, a process satisfies atomicity sphere if and only if (*i*) it does not contain any state that violates atomicity sphere (denoted by ϕ), and (*ii*) no non-retriable task is executed after a non-compensable task in any of its complete execution traces [16]. This is formalized as follows:

Definition 3 [Atomicity sphere criterion]: φ is an atomicity sphere predicate of a process, $\varphi(p)$ = *true* if and only if the following condition holds:

$\phi \notin p^* \wedge \neg\exists t \in trace(p)\ [(a=t[i],\ b=t[j],\ j>i>0) \wedge (C(a)=false,\ R(b)=false)]$

To check the atomicity sphere criterion in a service composition without disclosing the details of their backend process, atomicity-equivalent public views are derived from these backend processes, and are used to check the atomicity sphere instead of using the backend processes. This could be done by composing the atomicity-equivalent public views of all the involved services into a global process. If this global process satisfies atomicity sphere, then this service composition also satisfies atomicity sphere [16].

3. METHODOLOGY

3.1 Implicit Interactions and Their Impacts

As mentioned in Section 1, the atomicity property might still be violated in a service composition due to implicit interactions even if this composition satisfies atomicity sphere. In this section, we study the influence of implicit interactions on the atomicity property in a service composition, and propose methods to eliminate their impacts. Since atomicity is violated when a service composition needs to rewind and compensate a non-compensable task, implicit interactions may affect the atomicity of a service composition in the following two ways:

Abort-Dependency. The first way is illustrated in Figure 2(a) (which has been described in Section 1), where the execution of b_1 in service s_2 uses some result of a_2 in service s_1 (e.g., design document). If s_1 fails and compensates its task a_2, b_1 also needs rewinding and compensating, because the result of a_2 is no longer valid after a_2 has been undone. We refer to this kind of implicit

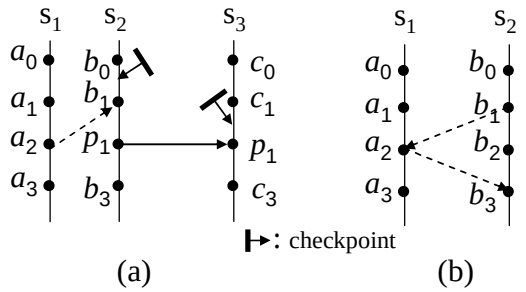

Figure 2. Impacts of implicit interactions and their handling.

interactions as *abort-dependency*. Note that we cannot model it as an explicit interaction at design time because there is no partial order between a_2 and b_1 in these two services.

Resource Conflict. The second way is depicted in Figure 2(b), where the execution of a_2 uses some result of b_1, and in turn, some result of a_2 is used by b_3. In this scenario, a_2 may cause b_3 fail because a_2 may change some result of b_1, which is required for correctly executing b_3. For example, suppose that b_1 buys some resources from a supplier, and b_3 uses these resources to produce product. b_3 may fail if these resources bought by b_1 are used up by a_2 and hence there are inadequate amount of these resources to complete b_3. In this case, service s_2 and its composition also need to rewind. The implicit interaction between a_2 and task pair b_1, b_3 is referred to as *resource-conflict*.

In the two above-mentioned cases, services s_1 may trigger service s_2 to rewind due to the implicit interactions. To ease the presentation, s_2 is referred to as an *affected service*. In practice, s_2 does not need to rewind to its initial state. Instead, it may just rewind to a *checkpoint*, where partial execution results are saved for subsequent service restoration. Let us assume that each checkpoint is owned wholly by an individual service and each service creates a checkpoint right after its initial state. When a constituent service in a service composition rewinds due to an implicit interaction, other services in this service composition may also need to rewind. These services then compensate all the committed tasks backwardly until their own checkpoint is reached and re-execute these tasks (or an alternative workflow) so as to eliminate the impacts of this implicit interaction. These tasks are referred to as *affected tasks* of this implicit interaction in this instance of the affected service. For example, as depicted in Figure 2(a), if a_2 needs compensating, s_2 and s_3 need rewinding to their checkpoints (e.g., the points between b_0 and b_1, c_1 and c_2, respectively), compensating all the committed tasks after these checkpoints and re-executing them. If some of these tasks are non-compensable (e.g., suppose c_3 has already committed and is non-compensable), atomicity is violated. This implicit interaction is said to be *afflicted*.

An intuitive way to avoid an abort-dependency or a resource-conflict implicit interaction is to restrict the concurrent execution of services. For instance, in Figure 2(a), we may choose to complete the service composition $s_2||s_3$ before starting s_1. However, it will compromise the service performance and interests of service providers (e.g., s_1 is unavailable until the service composition is completed). This is unattractive to the service provider. In addition, a web service may be advocated to be reusable in different service compositions. Avoiding the impact of implicit interactions by running the services one after another can reduce the extent of their reusability. Intuitively, it is more attractive to customize the

quality (e.g., atomicity) of a service to cater for the needs of different service compositions. Therefore, an alternative is to find the afflicted implicit interactions in each individual service composition and prevent them to occur in that composition.

In the following sections, we propose an approach to analyzing and locating the afflicted implicit interactions in a service composition. Measures can then be taken to eliminate their impacts on atomicity using these analysis results.

3.2 Locating Afflicted Implicit Interactions

To locate afflicted implicit interactions, one way is to verify all the potential implicit interactions in a service composition. However, it is hard to predict all the implicit interactions that may occur at runtime due to the open environment of web services.

We observe that instead of finding and verifying all the implicit interactions at runtime, we may locate those points in web services where implicit interactions may occur and evaluate their impacts on the atomicity property of a service composition. We further observe that, as mentioned in the previous section, an implicit interaction may occur whenever tasks in different services share resources. This motivates us to adapt the notion of data flow relations, in particular, *def-use* relation [1] to locate implicit interactions. Def-use is a kind of data flow analysis techniques to trace the definition of a data and its usage in program analysis. To analyze the potential implicit interactions, we apply the def-use techniques into services as follows:

Definition 4 [def-use]: Two tasks a and b satisfy the def-use relationship, denoted as *def-use*(a, b), if a can commit before b and b uses the data defined by a.

For instance, suppose a_2 and b_1 in Figure 3(a), b_1 and a_2, a_2 and b_3, b_3 and a_2, and b_1 and b_3 in Figure 3(b), all satisfy the def-use relationship. These relationships then represent the potential implicit interactions, e.g., the def-use relationship between a_2 and b_1 in Figure 3(a) represents the implicit interaction between a_2 and b_1. Note that identifying the def-use relationship amongst tasks is out of the scope of this paper. This can be done using traditional data flow analysis approaches [1]. We model the output of an intermediate result by a task as a data definition and the usage of an intermediate result by a task as a data usage. With the def-use relationship amongst tasks, these two kinds of implicit interactions are represented as follows:

Abort-dependency: Tasks a and b from different services (or service instances) form an abort-dependency implicit interaction, denoted as [a, b], if *def-use*(a, b), and compensating a needs compensating b.

Resource-conflict: Tasks a and b from the same service form a resource-conflict implicit interaction with another task c from other service (or instance), denoted as [a, c, b], if *def-use*(a, b), *def-use*(a, c) and *def-use*(c, b), and c may cause b to fail by changing the result of a required by b.

Note that the def-use relationships describe the syntax of implicit interactions and provides a mechanism to search all the potential implicit interactions systematically. Service providers could then filter the searching result based on application semantics. For example, if task b does not need compensating when a service compensates task a, then the [a, b] is removed from the set of potential abort-dependency implicit interaction. Similarly, if the supplier makes sure that committing any task c sharing the same resources with a and b in another service (or service instance) between the commitment of tasks a and b will not affect b, then the [a, c, b] could be taken out from the set of potential resource-conflict implicit interaction.

After identifying the potential implicit interactions a service may associate with, we proceed to locate the afflicted ones from them. To check statically whether an implicit interaction is afflicted, the key is to find out all its potentially affected tasks. A task is a *potentially affected task* of an implicit interaction if this task is an affected task of this implicit interaction in some particular instance of the affected service. It follows from our previous discussions that an implicit interaction is afflicted if the associated potentially affected task is non-compensable. To find out all the potentially affected task of an implicit interaction, we need to define a criterion. To do so, let us first define the criterion for a current checkpoint of a task. To ease the presentation, we represent a checkpoint as a specific task and use the predicate *checkpoint*(a) = *true* to represent that a is a checkpoint.

Definition 5 [Current Checkpoint Criterion]: Let t be a trace of a service s, i.e., $t \in trace(s)$. For any action $t[j]$, (j>0, and *checkpoint*($t[j]$) = *false*), a checkpoint cp is the current checkpoint for action $t[j]$ in the trace t, denoted as $cp = current(j, t)$, if $\exists i$: $0 < i < j$, $t[i] = cp$, $\wedge\forall l$ ($i < l < j$): $checkpoint(t[l]) = false$.

Intuitively, the current checkpoint defines the nearest restoration point of an action in a trace if this action fails. For example, the cp_1 is a current checkpoint for b_1 in service s_2 in Figure 3(a). Note that a communicating port action may have different current checkpoints in different services since it is shared by different services. For instance, in Figure 3(a), p_1 has two current checkpoints: cp_1 in the trace of s_2, and cp_2 in the trace of s_3.

In an instance t of a service composition, tasks may come from different involved services. Therefore, the calculation of current checkpoint for a task should use $t\backslash A_i$ instead of t, where A_i is the action set of the involved service for this task. To ease the presentation, we define a predicate $afterccp(t, m, n)$ to represent that $t[m]$ and $t[n]$ are from the same service s_j, and $t[m]$ is committed after the current checkpoint of $t[n]$ in the trace $t_j = t\backslash A_j$, that is, $afterccp(t, m, n) = true$ *iff* $\exists h>0, k > l >0$: $h = location(t, n, A_j)$, $k = location(t, m, A_j)$, $t_j[l] = current(h, t_j)$.

Definition 6 [Potentially Affected Task Criterion]: Given an implicit interaction δ (equal to [a, b] or [a, c, b]) that affects the service s_i in a service composition s ($=s_1\|\ldots\|s_n$), *PAT* is a predicate that evaluates the impact of this implicit interaction for any task in the service composition s. $\forall\gamma\in A$ (A is the action set of s), $checkpoint(\gamma)=false$: $PAT(\delta, \gamma)=true$, if and only if $\exists t\in trace(s)$: $r = t[g]$ ($g>0$), and $affected(\delta, t, g) = true$. Note that the predicate $affected(\delta, t, g) = true$ if and only if any of the following conditions is satisfied:

1) $\delta = [a, b] \wedge t[g] = b$
2) $\delta = [a, c, b] \wedge \exists n, m$: $n > g$, $n > m > 0$, $a = t[m]$, $b = t[n]$, $afterccp(t, g, n) = true$.
3) $\delta = [a, b] \wedge \exists k, l$: $k > 0$, $l > 0$, $b = t[k]$, $affected(\delta, t, l) = true$, $afterccp(t, g, l) = true$.
4) $\delta = [a, c, b] \wedge \exists n, m, l$: $n > g$, $n > m > 0$, $l > 0$, $a = t[m]$, $b = t[n]$, $affected(\delta, t, l) = true$, $afterccp(t, g, l) = true$.

Conditions 1 and 3 show that for an abort-dependency implicit interaction [a, b], task b is a potentially affected task because b is

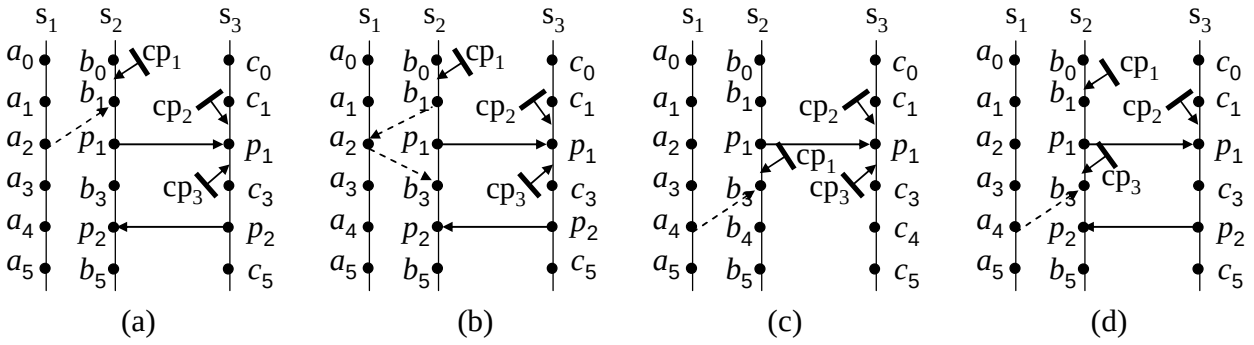

Figure 3. Affected tasks for implicit interactions.

affected by this implicit interaction. Moreover, if an action $t[l]$ from service s_j is marked as a potentially affected task in trace t of this service composition, then the tasks from service s_j committed after the current checkpoint of action $t[l]$ in trace $t\backslash A_j$ are also marked as potentially affected tasks. This is because these tasks also need compensating when rewinding service s_j from $t[l]$ to this checkpoint. For example, in Figure 3(a), b_1 is a potentially affected task for the implicit interaction $[a_2, b_1]$, and cp_1 is its current checkpoint. So, p_1, b_3, p_2, b_5 are all potentially affected tasks. Similarly, since p_1 is a potentially affected task and cp_2 in service s_3 is its current checkpoint. Therefore, tasks c_3 and c_5 are also potentially affected tasks. Note that not all the tasks committed after a potentially affected task in the service composition should be potentially affected tasks. For example, in Figure 3(c), b_3 is a potentially affected task of the implicit interaction $[a_4, b_3]$. Although c_3, c_4 and c_5 may have committed after b_3 in the service composition, they are not potentially affected tasks because they need not to be re-executed when we re-execute b_3.

Since the resource conflict implicit interaction may cause task b fail, Conditions 2 and 4 show that the tasks committed before b and after its current checkpoint in service s_i is potentially affected tasks. Moreover, if an action $t[l]$ from service s_j is marked as a potentially affected task in trace t of this service composition, then all the tasks from service s_j committed after the current checkpoint of action $t[l]$ and before the task b in trace t are also marked as potentially affected tasks. This is because these tasks also need compensating when rewinding service s_j from $t[l]$ to this checkpoint. For example, in Figure 3(b), p_1 is a potentially affected task for the implicit interaction $[b_1, a_2, b_2]$, and cp_1 is its current checkpoint. Therefore, b_1 is also a potentially affected task. However, b_3, p_2 and b_5 are not potentially affected tasks because they have not committed when b_3 fails. Similarly, since p_1 is a potentially affected task, and cp_2 in service s_3 is its current checkpoint, task c_3 should also be a potentially affected task but not p_2, c_5 because they commit after b_3 in the service composition.

This criterion is sound and complete for identifying potentially affected tasks of an implicit interaction in a service composition, as showed in the following theorem (and its proof is in [18]):

Theorem 1: Suppose an implicit interaction δ ($[a, b]$ or $[a, c, b]$) affects the service s_i in a service composition s ($= s_1 \| \ldots \| s_n$). $\forall \gamma \in A$ (A is the action set of s), γ is a potentially affected task of this implicit interaction, if and only if $PAT(\delta, \gamma) = true$.

Based on above analysis, we introduce the following algorithm to check whether a given implicit interaction is afflicted for a service composition s ($= s_1 \| \ldots \| s_n$) based on the atomicity-equivalent public views of all the other involved services. We represent a checkpoint by a special silent action τ_{cp}, which is compensable and retriable. Since this special silent action does not affect atomicity, its addition to a process does not affect the process's satisfaction of the atomicity sphere criterion. In this way, we can still use the atomicity-equivalent public view [16] to analyze the atomicity requirement in a service composition with checkpoints. The basic idea is to find out all potentially affected tasks, and check whether they are compensable. This algorithm constructs a potentially affected task set, *Pats*, to collect all such tasks in s for a given implicit interaction. The set is constructed in a recursive manner, based on the rules in Definition 6. Its correctness is proved in [18].

Algorithm 1 (Checking Afflicted Implicit Interaction):

Step 1: compose services $s_1,\ldots,s_n$ to form s [16].

Step 2: traverse the service composition s. For every trace t of s that contains some action $\beta = t[n]$ (and $\chi = t[m]$, $n > m > 0$) that forms implicit interaction $[\alpha,\beta]$ ($[\chi,\gamma,\beta]$),

Step 2.1: initialize the auxiliary set T_Pats to empty.

Step 2.2: add $t[n]$ ($t[l]$ where $afterccp(t, l, n) = true \wedge l<n$) to T_Pats

Step 2.3: for each $t[k] \in$ T_Pats,

Step 2.3.1: search t for the current checkpoint cp of $t[k]$.

Step 2.3.2: for every task $d = t[h]$ ($h>0$), if $afterccp(t, h, k) = true$ (and $h<n$) in trace t, add $t[h]$ to T_Pats.

Step 2.4: repeat Step 2.3 until T_Pats reaches a fix point.

Step 2.5: move all the tasks from T_Pats into Pats

Step 3: mark the implicit interaction $[\alpha,\beta]$ ($[\chi,\gamma,\beta]$) afflicted if and only if there exists one non-compensable task in Pats.

Let us illustrate this algorithm by the example in Figure 3(d). Suppose task b_1 is non-compensable, and to be identified as a potentially affected task. Initially, task b_3 is put into the empty auxiliary set *T_Pats* since this is an abort-dependency. Next, since cp_3 is the checkpoint of b_3, the tasks committed after cp_3 in service s_2 (i.e., p_2 and b_5) are added into *T_Pats* according to step 2.3. It then repeats step 2.3. As p_2 is in the set *T_Pats*, and p_2 has a current checkpoint cp_2 in service s_3. Therefore, in this round, tasks committed after cp_2 in s_3 (i.e., p_1, p_2, c_3 and c_5) are also added into *T_Pats*. By repeating step 2.3, it finds a new task p_1 in the set *T_Pats*, and p_1 has a current checkpoint cp_1 in service s_2. Hence, tasks committed after cp_1 in s_2 (i.e., b_1, p_1, b_3, p_2 and b_5) are added to the set *T_Pats*. It then repeats step 2.3 and finds that the set *T_Pats* does not change any more. Therefore, all the tasks in

T_Pats are potentially affected tasks, that is, b_1, p_1, b_3, p_2, b_5, c_3 and c_5. Since b_1 is non-compensable in this example, the algorithm identifies it from the set *Pats* and concludes that this implicit interaction is afflicted.

Note that when a service composition has loops, its traces may be infinite. However, intuitively, a trace only needs repeating a finite number of a composition's loops in determining potentially affected tasks (the set *T_Pats* reaches the fixed point). The proof is omitted here due to space limit (See [18]). Algorithm 1 could locate all the afflicted implicit interactions. Note that it is possible to terminate the algorithm when finding one non-compensable potentially affected task. However, we need the information of all the potentially affected tasks to suppress the impacts of this implicit interaction. Let us address that in the next section.

3.3 Behavior Constraint

As afflicted implicit interactions are caused by concurrent execution of services, an approach to suppressing the influence of afflicted implicit interactions is to impose some execution restriction to services. This may remove those afflicted implicit interactions or suppress their impacts on the atomicity property of a service composition. For instance, in Figure 3(b), we may forbid task a_2 to commit between the commitment of tasks b_1 and b_3. In this way, the implicit interaction will not occur, and this effectively suppresses its potential impacts on the atomicity property of the concerned service composition. To do so, we propose the use of behavior constraints for services:

Definition 7 [Behavior Constraint]: Let t_i be an instance of some service composition s_i, and f be a Boolean function. The first order logic formula $\forall t_1,\ldots,t_n, f(t_1, t_2,\ldots,t_n) = true$ is a behavior constraint over the set of service composition instances.

A behavior constraint quantifies a safety property amongst some service compositions when they are running concurrently. If executing a task will violate a behavior constraint, then the execution of this task will not occur until its execution will not breach the behavior constraint. If b_i and b_j from a service s_i and c from another service forms a resource-conflict implicit interaction causing atomicity violation in a service composition s of service s_i, the service provider of s_i may add the following behavior constraint to its services. To ease the presentation, we use predicate *commit(T, a)* to represent that task a has committed in a service instance T, *term*(T) to represent the termination action of service instance T (i.e., *abort* or *commit*), *ccp*(b) to represent the current checkpoint of action b in current instance. For two tasks a and b from instance T_i and T_j respectively, predicate *commit_before*(T_i, a, T_j, b) represents that T_i commits task a before T_j commits b.

Behavior constraint for resource-conflict implicit interaction (BC1): $\forall T_1 \in trace(s/s_i)$, $T_2 \in trace(s_k)$ $(T_1 \neq T_2)$: $[\neg\exists a \in A$: *def-use*$(b_i, b_j) \wedge$ *def-use*$(b_i, a) \wedge$ *def-use*$(a, b_j) \wedge$ *commit_before*$(T_1, b_i, T_2, a) \wedge$ *commit_before*$(T_2, a, T_1, b_j)]$, where s_k is another service provided by the service provider of s_i (may be equal to s_i).

As we model the commitment of a task as an action in our formal model, we further model a task to consume the required resources at the committing moment of this task. Therefore, the Boolean expression *def-use*$(b_i, b_j) \wedge$*def-use*$(b_i\ a) \wedge$ *def-use*$(a, b_j) \wedge$ *commit_before*$(T_1, b_i, T_2, a) \wedge$ *commit_before*(T_2, a, T_1, b_j) represents that task a has a resource conflict implicit interaction with b_i and b_j. Intuitively, this behavior constraint forbids any affecting task a from another service instance to commit between the commitment of tasks b_i and b_j in any instance of the service composition s. This effectively rules out any such afflicted implicit interaction in the service composition s. For example, in Figure 3(b), if task b_1 has already committed, but not b_3, then task a_2 will not commit until b_3 has committed. (Note that if task a_2 has committed before b_1 or after b_3, then it is not restricted by this behavior constraint).

For the abort-dependency implicit interaction, one may apply similar behavior constraints to suppress the afflicted ones. For example, in Figure 4(a), if task a_2 has already committed, then b_1 should not be committed until service s_1 terminates. However, this may reduce the performance of service s_2 because service s_1 may be long-running. An alternative approach is to use the information of potentially affected tasks to relax the behavior constraints.

For example, in Figure 4(a), the first behavior constraint is that, if none of the non-compensable potentially affected task (suppose the potentially affected tasks c_3 and b_5 are non-compensable in this example) has committed before task b_1, then b_1 is allowed to commit. However, the commitment of any of these non-compensable potentially affected tasks (that is, c_3 or b_5) of this implicit interaction $[a_2, b_1]$ is disallowed until service s_1 has terminated or has already committed at least one non-compensable task (suppose a_3 is non-compensable) after the current checkpoint of task a_2. This means that s_1 will not be rewound back to this checkpoint, compensate and re-execute task a_2 any more after the commitment of this non-compensable task (otherwise, atomicity of s_1 is violated). The second behavior constraint is that, as depicted in Figure 4(b), if some non-compensable potentially affected task (e.g., c_3) has already committed before b_3, then b_3 is not allowed to commit until s_1 has terminated or has committed a non-compensable task after the current checkpoint of task a_4. In this way, the atomicity property in the service composition of s_2 and s_3 is guaranteed and the behavior constraints are relaxed.

However, it is difficult to enforce the above two constraints because the affected tasks (e.g., c_3) of an abort-dependency implicit interaction may involve other remote services (which are autonomous and distributed) in the service composition. To alleviate this problem, we propose to propagate the non-compensability property of those potentially affected tasks to the service affected by the abort-dependency implicit interaction. The purpose is to use this property to enforce the local behavior constraints instead of using the potentially affected tasks from distributed services.

For example, as depicted in Figure 4(a), if we propagate the non-compensable property of the non-compensable potentially affected task c_3 to task p_1 (such that p_1 becomes non-compensable) in service s_2 (represented by the dashed arrow). Then we use task p_1 to substitute the role of c_3 in a behavior constraint. In this way, if a_2 commits before b_1, but no non-compensable task after the current checkpoint of a_2 has committed yet, then p_1 (so as c_3) is not allowed to commit until service s_1 has finished or already committed a non-compensable task after the current checkpoint of task a_2. In another situation, as depicted in Figure 4(b), the non-compensable property of task c_3 is propagated to the point after the checkpoint cp_1 (by inserting a non-compensable silent action). Then we use this inserted action to substitute c_3 in the behavior constraint. In this way, when task b_3 is ready to commit, this inserted non-compensable task has committed. If task a_4 has already committed, then task b_3 is not allowed to commit until service s_1 has terminated or already committed a non-compensable task after the current checkpoint of task a_4. The following rules summarize

above discussions on property propagation of the non-compensable potentially affected tasks.

Definition 8 [Property Propagation]: Suppose b from service s_i forms an abort-dependency implicit interaction $[a, b]$ affecting its service composition s (= $s_1 \| \ldots \| s_n$), and B is the set of its non-compensable potentially affected tasks. Let A_i, H_i be the set of actions and port actions of service s_i, respectively. The following rules propagate the non-compensable property of these tasks to service s_i, denoted as $s_i' = \Re(s_i)$:

Rule 1: $\forall t \in trace(s)$, if $\exists c \in B - A_i \wedge c = t[g] \wedge b = t[h]$ $(g > h > 0)$ $\wedge \exists d = t[f]$ $(h < f < g) \in H_i \wedge \neg\exists e = t[l]$ $(f < l < g) \in H_i$, then mark task d as non-compensable.

Rule 2: $\forall t \in trace(s)$, if $\exists c \in B - A_i \wedge c = t[g] \wedge b = t[h]$ $(h > g > 0)$, $\wedge \exists d = t[f]$ $(0 < f < g) \in A_i$, $\wedge \neg\exists e = t[l]$ $(f < l < g) \in A_i$, then insert a non-compensable silent action after d in service s_i.

Rules 1 and 2 represent the situations discussed in Figure 4(a) and (b), respectively. An algorithm based on Definition 8 could be derived straightforwardly. Due to space limit, this algorithm is omitted here. Note that the propagated non-compensable property in the service has no physical meanings (e.g., a compensable task c is changed to non-compensable by property propagation does not mean that it could not be compensated any more). This property is only used for behavior constraints and is only visible to the behavior constraints for the corresponding implicit interaction.

Based on the property propagation, we can use the local tasks in a service to define the two behavior constraints instead of the potentially affected tasks in another service. To represent these local tasks, the predicate $LPAT(c) = true$ if and if c is a local potentially affected task, or is a local task by property propagation (e.g., p_1 in Figure 4(a)). In this way, we formalize as follows the two corresponding behavior constraints for a given afflicted abort-dependency implicit interaction $[a, b]$ in a service s_i (after property propagation) for its service composition s. The correctness of the following two behavior constraints is proved in Theorem 2.

Behavior constraint for abort-dependency implicit interaction (BC2): $\forall T_1 \in trace(s/s_i)$, $T_2 \in trace(s_k)$, $(T_1 \neq T_2)$: $[[\exists c \in A: C(c) = false \wedge LPAT(c) \wedge def\text{-}use(a, b) \wedge commit\_before(T_2, a, T_1, b) \wedge (commit\_before(T_1, c, T_1, b) \vee b = c)] \Rightarrow [[\exists d \in A: C(d) = false \wedge commit\_before(T_2, ccp(a), T_2, d) \wedge commit\_before(T_2, d, T_1, b)] \vee commit\_before(T_2, term(T_2), T_1, b)]]$, where s_k is another service provided by the service provider of s_i (may be equal to s_i).

Behavior constraint for abort-dependency implicit interaction (BC3): $\forall T_1 \in trace(s/s_i)$, $T_2 \in trace(s_k)$ $(T_1 \neq T_2)$: $[[\exists c \in A: C(c) = false \wedge LPAT(c) \wedge def\text{-}use(a, b) \wedge commit\_before(T_2, a, T_1, b) \wedge commit\_before(T_1, b, T_1, c)] \Rightarrow [[\exists d \in A: C(d) = false \wedge commit\_before(T_2, d, T_1, c) \wedge commit\_before(T_2, ccp(a), T_2, d)] \vee commit\_before(T_2, term(T_2), T_1, c)]]$, where s_k is another service provided by the service provider of s_i (may be equal to s_i).

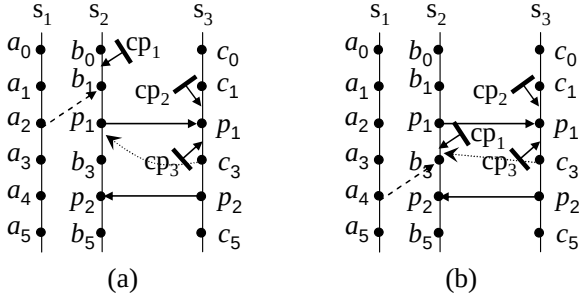

Figure 4. Property propagation of potentially affected tasks.

Theorem 2: Suppose task b from service s_i forms an afflicted abort-dependency implicit interaction $[a, b]$ in its service composition s (= $s_1 \| \ldots \| s_n$) with task a from another service composition s' (s' may be equal to s). Suppose s satisfies atomicity in the absence of s'. Let $s_i' = \Re(s_i)$, and BC_{i2}, BC_{i3} be the corresponding behavior constraints for this implicit interaction based on s_i'. $\forall T_1 \in trace(s)$, $T_2 \in trace(s')$ $(T_1 \neq T_2)$: if T_2 preserves atomicity property, and BC_{i2}, BC_{i3} are satisfied, then the implicit interaction $[a, b]$ will not cause atomicity violation for T_1.

This theorem can be proved directly based on Definition 8, BC2 and BC3. If all the involved services in a service composition are imposed local behavior constraints to handle their afflicted implicit interactions, and these behavior constraints are all satisfied, then the atomicity property of this service composition will not be violated by these implicit interactions (The proof is in [18]).

3.4 Implementation

In the previous section, we propose a behavior constraint approach to resolving the atomicity violation in service composition caused by implicit interactions. The behavior constraints could be enforced by different ways. In this section, we propose a simple implementation based on dynamic task scheduling.

The dynamic scheduling algorithm maintains a ready queue for tasks that are ready to commit in a service provider. The basic idea is to check whether the related behavior constraints are violated when committing a task. If there are violations, the task is not allowed to commit.

Algorithm 2 (Enforcing Behavior Constraints):

when the ready queue is not empty
get a task from the queue
for every behavior constraint
if the commitment of this task violates the constraint
then put the task back to the queue
if this task is not put back to the queue
then commit this task

Note that the checking of a constraint could use the information collected during the execution. This makes the checking take only O(1) time. For example, to check the behavior constraint 1, the execution information of task b_i, b_j could be collected during the execution of the service, whereas the def-use relationship could be indexed using hash table based on the shared resource. Note that this algorithm can be deployed on top of web service engines, such as BPEL engine, to enforce the behavior constraints.

4. EVALUATION

In this section, we illustrate our approach using two examples and then discuss the complexity and various limitations of our work.

4.1 Supply Chain Services

The first example is taken and adapted from [2], where there are two service providers, a supplier and a retailer, as depicted in Figure 5. The supplier has multiple services (e.g., s_1 and s_2) to provide their products to customers. The retailer orders products from the supplier based on its requirement.

Figure 5 describes a service composition of the supplier service s_2 and a retailer service s_3. The retailer first places an order to the

supplier, who then queries the stock to determine whether the required resources (e.g., mobile phone handsets) are available. If the warehouse does not have the required resources, service s_2 will purchase resources from others and put the resources in the warehouse (If no resource can be bought in time, then the supplier rejects the collaboration with the retailer). After order confirming, the supplier starts to produce the products for the retailer, and deliver them to the retailer. Meanwhile, the retailer starts to pre-sell the products to its customers and deliver the products to them after it receives the products from the supplier.

This collaboration will work well if the supplier does not execute other services that share the warehouse. However, when the supplier executes another service s_1 concurrently, this collaboration may fail due to implicit interactions. For example, suppose s_2 buys some resources and puts them into the warehouse. Then, another service s_1 queries the stock and uses up these resources to produce products in some other collaboration. As such, the task "*produce*" in service s_2 will fail because no resource is left and therefore its collaboration with s_3 fails. However, service s_3 may have pre-sold the products to its customers, and breach the sale contracts will incur nonetheless penalty to the retailer (e.g., loss of reputation and damage the public image of the retailer). Hence, atomicity property is violated in this situation.

This problem is caused by the resource conflict implicit interaction between services s_1 and s_2 (as marked by dashed arrows in Figure 5. Some other implicit interactions are not marked in this figure). One way to solve this problem is to avoid these implicit interactions by adding locks to service s_2 to prevent the resource from being used by other services during the duration between the tasks "*Query stock*" and "*Produce*". Since services are reusable components, adding locks to service s_2 will seriously compromise the performance of the supplier's services because the service s_2 may hold the locks for a long time in each of its collaboration. Moreover, this will increase the risk of denial of service attacks [5]. Therefore, this is unacceptable for the supplier.

Since not all collaborations with service s_2 would endure atomicity violation caused by these implicit interactions (e.g., if the retailer service does not pre-sell the products), a better way is to analyze the impacts of these implicit interactions on the atomicity in each of its collaborations and resolve only those violated ones. For example, using Algorithm 1, we can identify that these implicit interactions are afflicted in the service composition of s_2 and s_3 because this service composition has a non-compensable potentially affected task (i.e., "*pre-sale product*").

Based on the analysis result, the supplier can apply the behavior constraint (BC1) to service s_1 and s_2 to guarantee the atomicity property of this collaboration. In return, the supplier may want to charge the retailer for a higher price to use service s_2 in this collaboration because this collaboration will compromise the performance of other services provided by the supplier. This could be negotiated between the supplier and the retailer in their contracts.

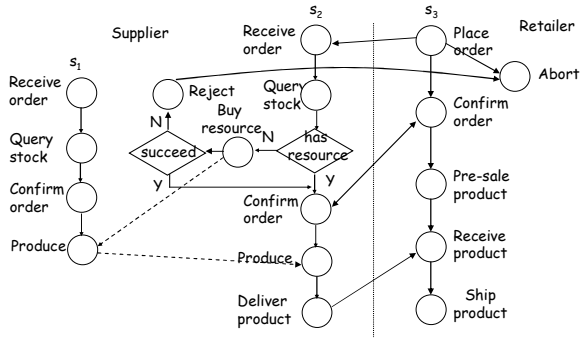

Figure 5. Service composition in supply chain.

4.2 Manufacturing Services

The second example is taken and adapted from [12], where there are two service providers, a manufacturer and a workshop, as depicted in Figure 6. The manufacturer supports two services: product design and product manufacturing. The workshop provides clients with a service for producing some customizable items. The manufacturer uses the workshop's service to manufacture their designed products.

In this example, to improve the efficiency, the manufacturer applies the pipeline techniques to design the products and produce them concurrently. The design service has an abort-dependency implicit interaction with the produce service by sharing the design documentation in the public database (that is, "*Write BOM*", and "*Read BOM*"). The produce service may compose services from different workshops to produce their customizable items.

In this situation, forbidding the implicit interaction between these two services by executing them one by one does not conform to the manufacturer's efficiency requirement (because the testing of a product design may take a long time, but the historical faulty rate for product design is low). On the other hand, if the testing reveals problems in the design of a product, then the design services need to fix the problems and update the design accordingly. As a result, the production service should stop the current manufacturing and use the updated design to manufacture the products. However, if the workshop service used by the produce service has committed some task whose effects are unable to compensate, then the atomicity is violated in the composition of the production service and the workshop service. For example, the task "*produce items*" may have committed and its effects are unable to compensate, because the produced items could not be used by others and become a waste (but they have consumed resources, e.g., parts and materials). Hence, the manufacturer may have to pay extra cost for these faulty products and endure the value loss.

One way to solve this problem is to avoid the implicit interaction between the design service and production service. However, as mentioned earlier, this is unacceptable for the manufacturer. Moreover, not all the collaboration between workshop services and production service has such value loss scenarios. Therefore, a better approach is to check the impact of implicit interactions for each service composition. For example, if the task "*produce*

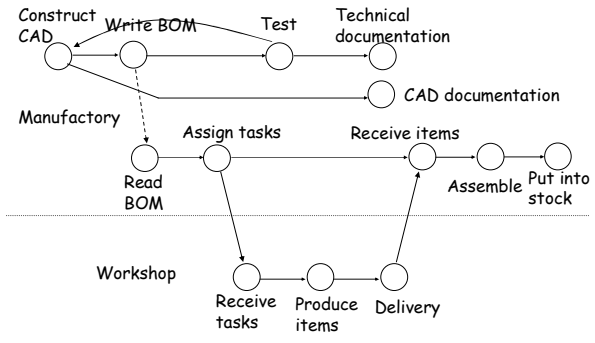

Figure 6. Service composition in manufacturing.

items" is non-compensable in a workshop service, then we can identify this implicit interaction as afflicted in their composition using Algorithm 1 (we can locate one non-compensable potentially affected task "*produce items*" for this implicit interaction).

The production service can choose to collaborate with another workshop service so that the implicit interaction does not affect their atomicity. An alternative is to apply our behavior constraint approach, that is, propagate the non-compensable property of task "*produce items*" to task "*Assign tasks*". With that, we only need to set behavior constraints (BC2 and BC3, details are available in [18]) for this service composition, which do not allow the task "*Assign tasks*" to commit until the design service finishes. In this way, the atomicity property is satisfied in this service composition.

4.3 Complexity Analysis

4.3.1 Locating Afflicted Implicit Interactions

Let k be the number of transitions of a service composition s (= $s_1 \| \ldots \| s_n$). When locating a potentially affected task in a trace, the algorithm wants to check if it commits after the checkpoint (and before task b if it is resource-conflict implicit interaction) in s. This takes at most $c \times k$ steps, where c is the maximum number of repeats for loops in s. Therefore, locating all the potentially affected tasks in a trace needs at most $c \times k^2$ steps. Let m be the number of traces s has (loops are not counted), the time complexity for Algorithm 1 is $O(m \times c \times k^2)$. According to the investigation of the processes from MIT process handbook [11], m and c are very small for these real life business processes. We will conduct more studies to confirm the scalability of Algorithm 1.

4.3.2 Property Propagation

Given a service composition s (= $s_1 \| \ldots \| s_n$), let k be the number of transitions s has. According to Definition 8, property propagation needs to traverse all the transitions of service composition s at most once. Therefore, the time complexity is $O(k)$.

4.3.3 Scheduling a task

In algorithm 2, scheduling a task needs checking all the behavior constraints (suppose the number is m) added by the service provider. Since checking one behavior constraint takes O(1) time, the time complexity of scheduling a task is $O(m)$. Currently, the scheduling algorithm prototype does not consider some specific scheduling requirements, such as real-time concerns, as our purpose is to illustrate the way to enforce the behavior constraints. This could be done by applying some advanced specific scheduling algorithms (e.g., real-time scheduling algorithm). A preliminary simulation study shows that the overhead of the scheduling is low (checking 20,000 behavior constraints totally only needs 280ms) [18]. We will further study the overall system performance using real life applications in the future.

4.4 Discussions

Conditional branch and nested structure. The current process model in our work does not support conditional branches, but the approach proposed in this paper can still work by assuming all the conditions are satisfied. In this case, the result of locating afflicted implicit interactions is conservative. This weakness is also the drawback of all static analysis approaches. It is possible to apply dynamic analysis approaches, such as symbolic execution technique [8], to refine the analysis results. However, such analysis leads to the state explosion problem. We will investigate how to combine static and dynamic approaches to analyze conditional branches in future. In addition, although the current model does not support processes with nested structure, this can be handled by unfolding the nested processes into a flat one.

Dynamic service composition. Currently, our work focuses on static service composition. When locating afflicted implicit interactions, we have the information about all the involved services. However, during the dynamic service composition, such information is incomplete, which makes the analysis difficult. We will investigate this issue in our future work.

Refining potentially affected tasks. When handling the impacts caused by an implicit interaction, it is possible to compensate and re-execute only those tasks that are really affected by this implicit interaction. For example, in Figure 3(a), if task c_5 does not use any value from task b_1 and its subsequent tasks, then we do not need to compensate and re-execute c_5 when services s_2 and s_3 rewind to their checkpoints. We will extend our work to support this by applying dataflow analysis in a service composition.

Handling implicit interactions. In this paper, we assume that service providers are willing to handle the impacts of implicit interaction in their services. An interesting question is, if they do not report the possible implicit interactions, or they do not handle implicit interactions, even if their services are affected by these implicit interactions, what will happen? This may impair the quality of their services. For example, in Figure 6, if the production service continues to use a faulty design to produce products, then the quality of the production service may be poor. In addition, service providers may choose to handle the afflicted resource-conflict implicit interaction using exception-handling approaches. In this case, they need to design their own exception handlers. However, doing so for an abort-dependency is non-trivial, because it is infeasible to predict when the services may be affected by the abort-dependency implicit interaction.

Deadlock problem. In this work, we propose to avoid or eliminate the impacts of implicit interactions by adding behavior constraints as safety properties in each service provider. This may cause deadlock problems between different service compositions. This problem can be alleviated by using existing works (e.g., [9]) on detecting and resolving deadlocks in distributed systems.

Other implicit interactions. In this work, we discuss two general implicit interactions that may affect the atomicity property of a service composition. Other specific application semantics (e.g., a shared document will be invalid after expire day) may also cause processes rewind. The detection and resolution of such specific afflicted implicit interactions could be handled in a similar way.

5. RELATED WORK

Let us here review the major techniques proposed by recent studies in the areas of transactional support for service computing, data races in concurrent systems, fault-tolerant systems, and make a comparison with our work.

Recently, some transactional protocols are proposed for service compositions. Business Transaction Protocol (BTP) [3] meets the requirements for long running collaborative business applications. It uses a two-phase outcome protocol to guarantee the atomicity property amongst multiple participants by requiring them to commit their tasks together. Another protocol, WS-T [15], defines two transaction models (one for short duration and another for long duration) to support atomicity in service compositions. How-

ever, these works do not address the impacts of implicit interactions on the atomicity property.

Many researchers also studied the isolation issue for transactional workflow and service computing. Schuldt et al. [12] addressed the recoverability problem based on the serializability of processes, which, however, is quite restrictive because this will compromise their performance due to their long running nature and is therefore unacceptable in service computing [2][5][13]. As a result, some researchers proposed to relax the serializability criterion using application semantics [2][13]. These approaches could handle the resource-conflict implicit interactions. The difference is that they address this issue from the perspective of completing a transaction. If a resource conflict does not change the outputs of the application at the end, then such an interaction, according to these works, need not be handed. However, this resource conflict may still lead to atomicity violation 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 complements theirs. In addition, the impact of abort-dependency on atomicity property in a service composition has not been addressed in these works.

Many works on detecting and preventing data races in concurrent systems have been proposed (e.g., [4][14]). The resource conflict may be regarded as a kind of data race. However, these approaches are incompetent for service-oriented applications because of the autonomous and heterogeneous nature of web services. Moreover, different from concurrent programs, not all data races in service compositions are regarded as errors. Our work focuses on locating only those that might affect the atomicity property in a service composition.

Another well-studied area is distributed, cooperative fault-tolerant systems [10], where consistency is achieved by rolling the execution back to some checkpoints using backward recovery and restarting the execution from the checkpoints in case of a failure. This mechanism is similar to our approach. The difference is that instead of simply rolling back the systems to their checkpoints, service recovery needs to compensate the effects of committed tasks, some of which, however, may be non-compensable. Therefore, we need to detect the potential atomicity violation scenarios and preclude them in advance.

6. CONCLUSION

This paper presents a novel approach to detecting and resolving the atomicity violation in service compositions caused by implicit interactions among these composed services. By using the def-use relationship amongst tasks, potential implicit interactions are identified. An algorithm is then proposed to locate the afflicted implicit interactions. Based on these analysis results, we propose to introduce behavior constraints to each involved service as local safety property in a service composition. We show that the satisfaction of these safety properties exempts the atomicity property in service compositions from being violated by implicit interactions. The implementation of behavior constraints is also proposed and the complexity of our algorithm is analyzed.

The present work is still subject to limitations, e.g., not supporting dynamic service composition and process evolution. In the future, we will address these issues and conduct more real-life case studies to understand the applicability and scalability of our approach in practice.

7. ACKNOWLEDGMENTS

This research was supported by grants from the Research Grants Council of Hong Kong (Project No. HKUST6167/04E) and the National Natural Science Foundation of China (Grant No. 60673112). The authors would also like to thank the anonymous reviewers for their constructive comments and suggestions.

REFERENCES

[1] Allen, F.E., and Cocke, J. A program data flow analysis procedure. *CACM*, vol.19, no.3, March 1976, pp. 137–146.

[2] Arpinar, I.B., Halici, U., Arpinar, S., and Dogac, A. Formalization of workflows and correctness issues in the presence of concurrency. *Distributed & Parallel Databases*, vol.7, no.2, Apr 1999, pp. 199–248.

[3] BTP. Available at: http://www.oasis-open.org/committees/tc_home.php (accessed on March 7, 2007).

[4] Choi, J-D, Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. of PLDI*, May 2002, pp. 258–269.

[5] Fekete, A., Greenfield, P., Kuo, D., and Jang, J.L. Transactions in loosely coupled distributed systems, In *Proc.of ADC*, Adelaide, South Australia, Feb. 2003, pp. 7–12.

[6] Greenfield, P., Kuo, D., Nepal, S., and Fekete, A. Consistency for web services applications, In *VLDB*, Trondheim, Norway, Aug-Sept. 2005, pp. 1199–1203.

[7] Hagen, C., and Alonso, G. Exception handling in workflow management systems. *IEEE TSE*, vol.26, no.10, Oct. 2000, pp. 943–958.

[8] King, J.C. Symbolic execution and program testing. *CACM*, vol.19, no.7, July 1976, pp. 385–394.

[9] Kshemkalyani, A.D., and Singhal, M. Efficient detection and resolution of generalized distributed deadlocks. *IEEE TSE*, vol.20, no.1, Jan. 1994, pp. 43–54.

[10] Lee, P.A., and Anderson, T. *Fault tolerance: Principles and practice*, Springer-Verlag 1990.

[11] MIT process handbook online, available at: http:// process.mit.edu/Info/CaseLinks.asp (accessed on June 7, 2007)

[12] Schuldt, H., Alonso, G., Beeri, C., and Schek, H.J. Atomicity and isolation for transactional processes. *ACM TODS*, vol.27, no.1, Mar. 2002, pp. 63–116.

[13] Waechter, H., and Reuter, A. The ConTract model. In *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992, pp. 219–263.

[14] Wang, L., and Stoller, S.D. Runtime analysis of atomicity for multithreaded programs. *IEEE TSE*, vol.32, no.2, Feb. 2006, pp. 93–110.

[15] WS-T. Available at: http://msdn.microsoft.com/ws/2002/08/wstx/ (accessed on March 7, 2007).

[16] Ye, C.Y., Cheung, S.C., and Chan, W.K. Publishing and composition of atomicity-equivalent services for B2B collaboration. In *Proc. of ICSE*, 2006, China, pp. 351–360.

[17] Ye, C.Y., Cheung, S.C., Chan, W.K, and Xu, C. Local analysis of atomicity sphere for B2B collaboration. In *Proc. of FSE*, Nov. 2006, Portland, Oregon, USA, pp. 186–196.

[18] Ye, C.Y., Cheung, S.C., Chan, W.K, and Xu, C. Atomicity violation analysis for service composition. Technical report, No.HKUST-CS-07-03, Depart. of Comp. Sci. & Eng, The Hong Kong University of Science & Technology, 2007.