JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1565

An approach for mining service composition patterns from execution logs

Bipin Upadhyaya, Ran Tang and Ying Zou*,†

Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada

ABSTRACT

A service-oriented application is composed of multiple Web services to fulfill complex functionality that cannot be provided by individual Web service. The combination of services is not random. In many cases, a set of services are repetitively used together in various applications. We treat such a set of services as a service composition pattern. The quality of the patterns is desirable because of the extensive uses and testing in the large number of applications. Therefore, the service composition patterns record the best practices in designing and developing reliable service-oriented applications. The execution log tracks the execution of services in a service-oriented application. To document the service composition patterns, we propose an approach that automatically identifies service composition patterns from various applications using execution logs. We locate a set of associated services using Apriori algorithm and recover the control flows among the services by analyzing the order of service invocation events in the execution logs. We also identify structurally and functionally similar patterns to represent such patterns in a higher level of abstraction regardless of the actual services. A case study shows that our approach can effectively detect service composition patterns. Copyright © 2012 John Wiley & Sons, Ltd.

Received 28 February 2011; Revised 2 May 2012; Accepted 5 June 2012

KEY WORDS: component; Web service; composition; pattern mining; log

1. INTRODUCTION

A service-oriented application is composed of multiple Web services to provide complex functionality that cannot be achieved by a single Web service. Service-oriented architecture (SOA) developers discover [1] desired Web services and compose them into a service-oriented application using specification languages, such as Web Services Business Process Execution Language [2]. A Web service contains many operations. An operation provides services to a client. An operation is invoked through a remote procedure call using the simple object access protocol (SOAP). In this paper, we call an operation in a Web service as a service. Different service-oriented applications can be composed to fulfill the similar functional requirements from various organizations. For example, a travel reservation application may contain a Web service to reserve tickets for sports events in the destination. Another travel reservation application may use a Web service to search for nearby restaurants. Variations exist in the two different applications. However, most of travel reservation applications deliver some common functionality, such as booking transportation and accommodation. The set of Web services delivering such common functionality can be identified as a service composition pattern. In general, a service composition pattern consists of a set of operations (i.e., services) from different Web services and the control flows among the operations.

*Correspondence to: Ying Zou, Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada.
† E-mail: ying.zou@queensu.ca

The control flow defines the execution order in which the services should be invoked (e.g., sequential order, parallel order, or alternative order). The services contained in a pattern are frequently executed together by service-oriented applications in the defined execution order.

A service composition pattern captures the expert knowledge in designing service-oriented applications. Similar to the concept of the software design patterns provided by Gamma *et al*. [3], a service composition pattern documents a recurring solution to the same problem that happens over and over again. Service composition patterns reflect the best practices. The solution conveyed in a service composition pattern can be reused in different applications to solve the same problem without reinventing the wheel. In particular, they are well used by a large amount of adoptions. Therefore, a service-oriented application potentially has better quality when it is composed by reusing service composition patterns. Service composition patterns can help to improve the functionality and quality of existing applications. By comparing an existing application against the patterns, a maintainer can identify the discrepancy in terms of the functionality and the quality, and consider applying the patterns in the application. Moreover, Web services used in patterns are more heavily used than other Web services. Thus, the patterns can be used to optimize the allocation of maintenance personnel and resources. Service maintainers can allocate more resources to maintain and improve the Web services used in a pattern.

To improve the quality of service-oriented applications, a great amount of research effort has been devoted in identifying well-accepted service composition patterns for reuse in the service composition. The research can be divided into top-down and bottom-up approaches. In the top-down approaches [4], the business process management architects compare the business processes from different organizations to identify commonalities and document them as patterns. However, this static approach does not consider the frequency of executing the applications when identifying the patterns from high-level business process models. In the bottom-up approaches [5–11], the execution logs of applications are analyzed to mine business processes. However, the existing research focuses on recovering the control flows of a single business process without extracting patterns shared among multiple applications.

To identify patterns frequently used in practice, we present an approach that extends existing bottom-up approaches to mine service composition patterns from execution logs of service-oriented applications. Instead of recovering a single business process from the logs, we identify frequently executed patterns used by multiple service-oriented applications. To facilitate the reuse of a service composition pattern as an independent and ready-to-use component, we further infer the control flows within the pattern. Some of the patterns discovered from execution logs may convey the similar functionality with the similar control flow structure. However, the actual services used in such patterns may be different. We further analyze the functional and structural similarity among the patterns and represent the similar patterns in a higher level of abstraction independent of the actual Web services. This paper extends earlier work [12] and [13]. We enhance the earlier work in the following aspects:

1. We propose an approach to identify the functional and structural similarities among patterns recovered from execution logs. The discovery of functionally and structurally similar patterns allows us to present the patterns in a higher level of abstraction that focuses on the functionality of services and their control flows.
2. We extend the existing case study to include two additional domains from medical and telecommunication. We further validate the benefits of our approach through the extended case study.
3. We conduct a case study to validate our approach to identify functionally similar patterns.

The remainder of this paper is organized as follows. Section 2 gives an overview of our approach. Section 3 presents an approach to identify a set of services commonly used together. Section 4 discusses an approach to identify control flows among the services in a pattern and patterns with functional and structural similarities. Section 5 presents our case study. Section 6 discusses the related work. Finally, Section 7 concludes the paper and explores the future work.

2. OVERVIEW OF OUR APPROACH

Figure 1 gives an overview of our approach, which is broken down into four major steps: (i) collecting and preprocessing execution logs; (ii) identifying frequently associated Web services;

DOI: 10.1002/smr

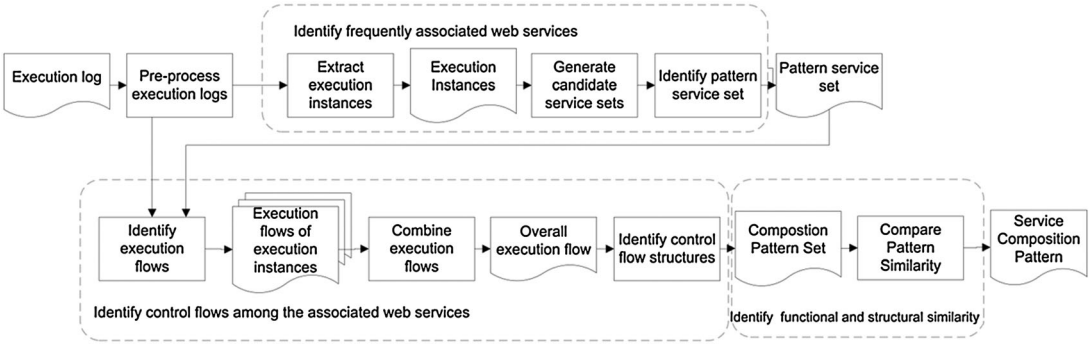


Figure 1. Overview of our approach.

(iii) recovering the control flow among the services identified as a pattern; and (iv) identifying functionally similar patterns.

2.1. Collecting and preprocessing execution logs. The composed application is deployed and executed in a runtime execution environment, such as IBM WebSphere Process Server [14]. Traditionally, applications are individually deployed on a proprietary infrastructure owned by the organization that operates and utilizes the application. With the increasing adoptions of "Platform as a service" paradigm, which provides a centralized runtime execution environment, more and more service-oriented applications are deployed on centralized runtime execution environments, such as Microsoft Azure Services Platform [15] and Google App Engine [16]. For instance, a platform named Heroku [17] hosts over 45,000 applications in 2007. Compared with the traditional deployment model, "Platform as a service" paradigm can reduce the cost and the complexity of managing the underlying hardware and software. Adoptions of such a paradigm facilitate the monitoring of the execution of a large number of services-oriented applications to obtain the execution logs.

Once a service-oriented application is deployed in a runtime execution environment, the application can be executed in many execution instances. Execution instances from different applications may co-exist. In each execution instance, events can be triggered. We record the triggered events in the log using the logging facility provided by the execution environment. An execution log contains different types of events. For example, resource adapter events record the interaction between a service-oriented application and a legacy system. Business rule events track the runtime status of business rules. Service invocation events indicate the timeline of a Web service execution. We are interested in instrumenting service invocation events. In particular, an ENTRY event is triggered when an operation (i.e., service) of a Web service is invoked. An EXIT event occurs when a service completes the computation and returns results. Each event is recorded with the time of triggering, the name of the service that triggered the event, the id of the execution instance, and the underlying application.

Figure 2 shows the format of the events logged by the IBM WebSphere Process Server. The *timestamp* in Figure 2 is a fully qualified date (i.e., YYMMDD), 24 h time with millisecond precision, and the time zone. Each execution instance is uniquely identified with an identifier (i.e., <*Id*>). *Id* is an eight-character hexadecimal value generated from the hash code of the thread that issued the trace event. *shortName* in Figure 2 is the name of the component. *eventType* indicates the type of an event generated. The ENTRY/EXIT event is recorded in this field. The *classname* and *methodname* are the Web service and the operation that generate the message (i.e., *textmessage*). *Parameters* record the parameter names and the values used in the event. The parameters show the data flows between the execution instances of Web services. All

```
<timestamp><Id><shortName><eventType>[className][methodName]<textmessage>
        [parameter 1]
        [parameter 2]
```

Figure 2. Format of events recorded.

DOI: 10.1002/smr

the events are captured in the execution logs. We need to process the logs to extract the service invocation events (i.e., ENTRY and EXIT events).

2.2. Identifying frequently associated Web services. Each execution instance invokes a set of services. To detect a service composition pattern, we find a set of associated services that frequently appear together in many execution instances. Such frequently associated services form a service set for a service composition pattern (i.e., pattern service set). The number of services in a service composition pattern is important. From the result of our case study, we identify a service composition pattern with more than four services (i.e., threshold = 5). We count the number of task (nodes) for each application and find that less than five services do not provide a reusable functionality in our subject applications. However, the required number of services may vary in different settings. We generate the candidate service sets and select the one with the highest frequency as the intermediate pattern service set. From the intermediate pattern service set, we construct the final pattern service set.

2.3. Recovering the control flow among the services identified as a pattern. The control flow among services would make a pattern more structured and ease the reuse of the pattern as an independent component in the service composition. To recover the control flow of a pattern, we use the execution instances that contribute to the pattern. For each execution instance, we identify the execution flow among the services in the pattern. We combine execution flows from all execution instances to obtain the overall execution flows of the service composition pattern. We further infer the control flow structures (e.g., sequential structure and parallel structure) from the overall execution flows.

2.4. Identifying functionally similar patterns. Different departments of an organization may use different service providers for fulfilling the same business goal. Moreover, an organization may decide to change service providers as the result of strategic or operational decisions. For example, an organization previously uses UPS for product shipment. Later, it decides to use DHL for Asian region and UPS for other regions. As a result, our approach would recover more than one functionally similar pattern. The recovered patterns use different services to accomplish the same business objective. To represent the recovered patterns in a higher level of abstraction regardless of the actual services used, we infer functional and structural similarity among the recovered patterns. The functional similarity among services is measured by the similarity in the names of the operations, input parameters, and output parameters between the involved services. Two patterns are considered to be structurally similar if the control flow of one pattern is present in the control flow of the other.

3. IDENTIFICATION OF FREQUENTLY ASSOCIATED SERVICES

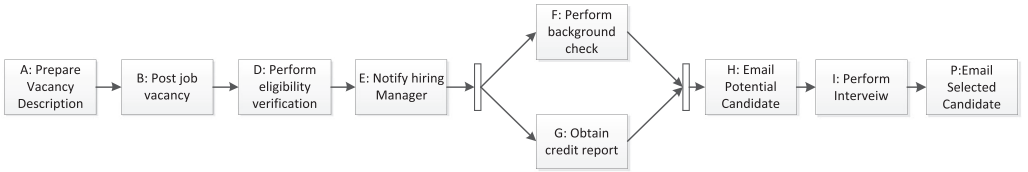
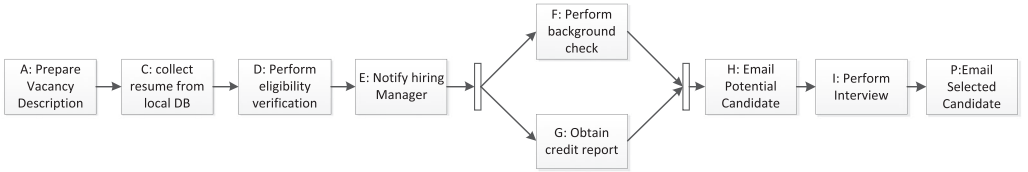
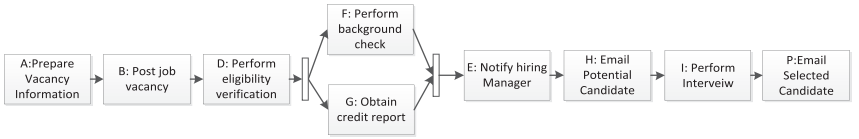
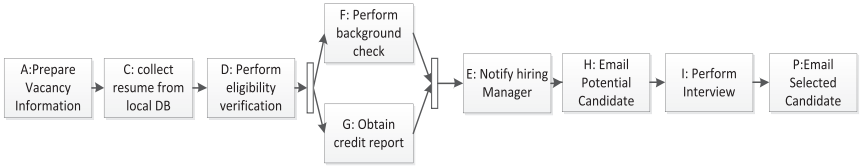
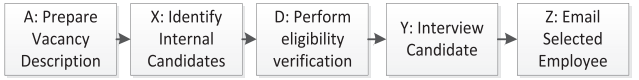
In this section, we discuss our approach to identify a set of frequently associated services (i.e., a pattern service set). In our approach, we use the Apriori [18] algorithm and develop a set of heuristic rules.

3.1. Extracting Execution Instances from Execution Logs

We analyze the execution logs to extract execution instances that may belong to different applications. Each execution instance contains a set of services. Using the execution instance id recorded in the events, we collect all the events triggered by the same execution instance into one group. We extract the names of services recorded in the events. As a result, we can obtain the services invoked in each execution instance. For example, Table I shows eight execution instances corresponding to the execution of four different service-oriented applications. Table I also presents the business process of each instance. The four applications in the example are used in the recruitment process in the Human Capital Management domain. The business processes in execution instances 1, 2, 3, 4, 7, and 8 are related to the process of recruiting new employees. Business processes in execution instances 5 and 6 describe the steps for the internal promotion of employees. Each letter represents a service. In a real application environment, the Uniform Resource Identifier (URI) of a service is unique to each service provider, and hence, it is used to differentiate services. It is also possible that a service is invoked more than once (e.g., in a loop shown in instances 7) in one execution instance.

DOI: 10.1002/smr

Table I. Example execution instances.

Execution instances ID	App ID	Business process	Services contained in the execution instance
1	1		{A, B, D, E, F, G, H, I, P}
2	1		{A, C, D, E, F, G, H, I, P}
3	2		{A, B, D, F, E, G, H, I, Q}
4	2		{A, C, D, F, E, G, H, I, Q}
5	3		{A, X, D, Y, Z}
6	3		{A, X, Y, D, Z}

(Continues)

DOI: 10.1002/smr

Table I. *Continued*

Execution instances ID	App ID	Business process	Services contained in the execution instance
7	4		{A, B, D, E, F, G, D, E, F, G, H, I, S}
8	4		{A, C, D, E, F, G, D, E, F, G, H, I, S}

DOI: 10.1002/smr

The multiple occurrences of a service should not affect the technique because we do not consider the number of occurrences or the order of services to determine the associated services.

3.2. Generating Candidate Service Sets

We enumerate the possible combinations of services with varying sizes to form candidate service sets. From the candidate service sets, we construct the pattern service set that contains the services in a service composition pattern. More specifically, we generate the candidate service sets starting from size 1 and then increasing to size 2 and so forth. A service that appears in at least one execution instance becomes a candidate service set of size 1. We use "support" to measure the frequency of a candidate service set. A support value counts the number of execution instances that invoke all the services in a candidate service set. For the example shown in Table I, the candidate service set, {D}, is invoked by eight execution instances. Therefore, the support of {D} is 8. The supports of all candidate service set of size 1 are shown in Figure 3(a).

Given the candidate service sets of size n, we generate candidate service sets of size $n+1$ in the following steps:

1. Collect a candidate service set that contains the first $n-1$ identical services and only one different service to form one group. For the example shown in Figure 3(b), the size of each candidate service set is 2 (i.e., $n=2$). We group the candidate service sets, {A, B}, {A, C}, {A, D} and {A, F}, together because of their first identical (i.e., $n-1$) service, A.
2. Expand candidate service sets of size n to size $n+1$. In the new candidate service sets, the first $n-1$ services are same as the initial sets of size n. The nth and $(n+1)$th services are pair-wise combinations of the nth services of the initial set of size n. For the example shown in Figure 3(b), the second services in Group A are B, C, D, and F. The pair-wise combinations of the second service are BC, BD, BF, CD, CF, and DF. These combinations become the second and third services for the new candidate service sets. As a consequence, the new candidate service sets of size 3 are {A, B, C}, {A, B, D}, {A, B, F}, {A, C, D}, {A, C, F}, and {A, D, F}.
3. Calculate the support of each new candidate service set. For example, the supports of a new candidate service set of size 3 are listed in Figure 3(c).
4. Filter out infrequent candidate service sets. The number of possible candidate service sets can grow exponentially when the size of candidate service sets increases. To improve the efficiency in identifying the most frequently occurred candidate service sets, we filter out the infrequent candidate service sets of size n to reduce the number of candidate service sets of size $n+1$. We define a threshold support shown in equation (1). The threshold support value is the average number of execution instances of a service-oriented application. If a candidate service set is more frequently executed than the average frequency, then such a candidate service set can be qualified as a pattern service set. We filter out the candidate service sets with the support less than or equal to the threshold value. For the example shown in Table I, the threshold support is 2 (i.e., 8/4). Hence, a candidate service set with the support less than or equal to 2 (i.e., support ≤ 2) is filtered out as shown in Figure 3.

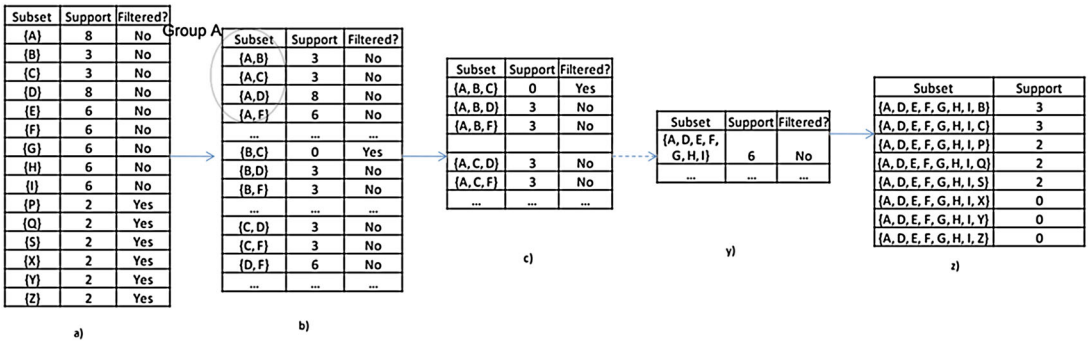


Figure 3. Example of generating candidate service sets.

DOI: 10.1002/smr

$$\text{threshold} = \frac{\text{execution instances}}{\text{service-oriented apps}} \tag{1}$$

5. Repeatedly generate larger candidate service sets until no further candidate service sets can be achieved. Specifically, we record the highest support value (e.g., H) achieved by candidate service sets of size 4. Any candidate service set of size greater than 4 should have a support value no more than H. If no candidate service set with size m (i.e., $m > 4$) can achieve the support value H, there is no need to further expand from candidate service sets of size m to candidate service sets of size $m+1$ because further expansion will not achieve any candidate service sets with a support value H. Hence, we stop the expansion. We rank all the candidate service sets with size greater or equal to 4 on the basis of their supports. We select the one with the highest support (i.e., H) as the intermediate pattern service set S. If more than one candidate service sets have the highest support, we choose the one with the most services. For the example shown in Figure 3, the candidate service set, {A, D, E, F, G, H, I}, is selected as the intermediate pattern service set S, which indicates that services A, D, E, F, G, H, and I are frequently executed together.
6. Identify a larger pattern service set by merging the candidate service sets that contain a common subset (i.e., the intermediate pattern service set S in step 5) and an additional service. For example, eight candidate service sets containing S are shown in Figure 3(z) with support 3, 2, and 0. In this step, we handle alternative structures in the control flow of a service-oriented application as shown in Figure 4. Suppose we have the candidate service sets, S1 (i.e., $S1 = S \cup \{\text{Service1}\}$) and S2 (i.e., $S2 = S \cup \{\text{Service2}\}$). We use S_{merged} (i.e., $S_{\text{merged}} = S \cup \{\text{Service1, Service2}\}$) as the final pattern service set. We merge service sets S1 and S2 if the following two conditions are met:

   1. S1 and S2 are executed by the same set of applications.
   2. The support of the four service sets (i.e., S, S1, S2, and S_{merged}) satisfies equation (2).

$$support(S) = support(S1) + support(S2) - support\left(S_{\text{merged}}\right) \tag{2}$$

Condition (2) indicates that Service1 and Service2 are likely in the two execution branches of an alternative control flow structure. When the alternative structure is not within a loop, either Service1 or Service 2 is executed in one execution instance. In this case, support (S) is the sum of support (S1) and support (S2). When the alternative structure is within a loop, both Service1 and Service2 can be executed in two or more iterations in a single execution instance. Therefore, we need to reduce support (S_{merged}) from the sum of support (S1) and support (S2) to avoid counting of such execution instances more than once.

For example, {A, D, E, F, G, H, I} in Figure 3(y) is an intermediate pattern service set S in step 5. We further check its expanded candidate service set in Figure 3(z). We find that the sum of the support for {A, D, E, F, G, H, I, B} (i.e., S1), and {A, D, E, F, G, H, I, C} (i.e., S2), is 6. The support of S_{merged} (i.e., {A, D, E, F, G, H, I, B, C}) is 0. The support of S (i.e., {A, D, E, F, G, H, I}) is 6. The condition (2) is met. Moreover, S1 and S2 are both executed by App1, App2, and App4. The condition (1) is satisfied. We use S_{merged} (i.e., {A, B, C, D, E, F, G, H, I}) as the final pattern service set. The final pattern service set contains *Prepare vacancy Description*, *Post Job Vacancy*, *Collect Resumes from*

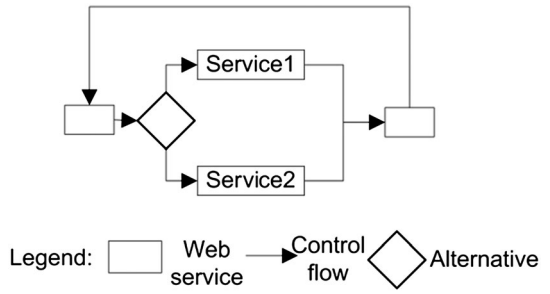


Figure 4. Alternative structure in a service-oriented application.

DOI: 10.1002/smr

Local DB, *Perform Eligibility Verification*, *Notify hiring Manager*, *Perform Background Check*, *Obtain Credit Report*, *Email Potential Candidate*, and *Perform Interview*.

Using the aforementioned steps, we identify a final pattern service set. We also collect the execution instances that contribute to the pattern. Specifically, if no merging is achieved in step 6), we use S as the final pattern service set, and the execution instances containing S are identified as the execution instances that contribute to the pattern. If merging is achieved in step 6), the final pattern service set is S_{merged}, and the execution instances containing S1, S2, or S_{merged} are identified as the execution instances that contribute to the pattern.

4. RECOVERY OF CONTROL FLOWS OF A SERVICE COMPOSITION PATTERN

In this section, we discuss our approach to recover the control flow among services in a service composition pattern. An execution flow is a graph in which the nodes represent services and the edges denote service ordering relations. Examples of execution flows for execution instances can be found in Table II. We combine the execution flows of execution instances to obtain the overall execution flows of the pattern. We further infer the control flow structures from the overall execution flow.

Figure 5 illustrates the execution instances that contribute to the pattern as identified in the last section. Instances 5 and 6 do not contribute to the pattern, and they are not included in Figure 5. The service invocation events are also shown in the figure. Such execution instances initially contain other services (as shown in Table I) that do not belong to the final pattern service set. We disregard such services.

Table II. Examples of the identified execution flow for each execution instance.

Instance ID	Execution flow
1	A → B → D → E → F → G → H → I
2	A → C → D → E → F → G → H → I
3	A → B → D → {F, E} → G → H → I
4	A → C → D → F → E → G → H → I
7	A → B → D → {F, E} → G → H → I
8	A → C → D → {F, E} → G → H → I

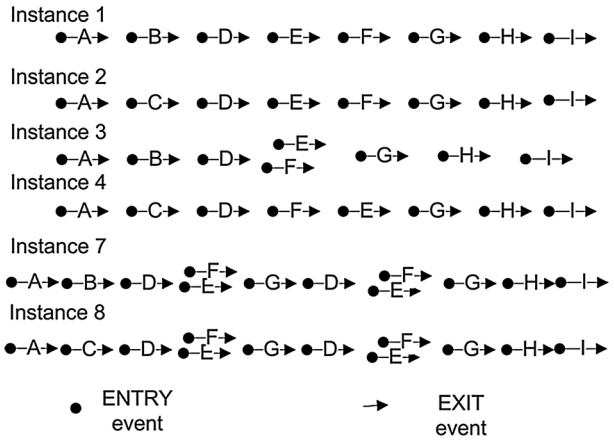


Figure 5. The execution instances with the timelines of service invocation events.

DOI: 10.1002/smr

4.1. Identifying Execution Flows of an Execution Instance

We identify three ordering relations among services in an execution flow: precedence relation, iterative relation, and parallelism relation.

4.1.1. Precedence Relation. A precedence relation connects a service to its successor service and represents that the service executes immediately before the successor. Precedence relation is denoted by a directed edge from service A to the successor service B (i.e., $A \rightarrow B$) in the execution flow graph.

To ensure that precedence relation is identified between two services A and B only when service B is executed immediately after service A, two conditions need to be satisfied: (i) the ENTRY B event occurs after the EXIT A event; and (ii) no services can start and complete its execution between the EXIT A event and the ENTRY B event.

To identify service ordering relations, we can examine each pair of services and check if the services meet the aforementioned two conditions. However, it may be a time-consuming process. To ease the identification of the service ordering relations, we build an event graph to describe the ordering among events triggered by each execution instance. In an event graph, a node represents an ENTRY event or an EXIT event. For example, Figure 6 shows an event graph constructed using instance 7 from Figure 5. If a service is invoked more than once, it has more than one ENTRY event and EXIT event. We treat these events as distinct events and create a node for each of them. A directed edge from node A to node B represents that event B immediately follows event A in the execution instance.

A precedence relation is represented as a precedence edge that emanates from an EXIT event node and is incident on an ENTRY event node. For the example shown in Figure 5, the precedence relation between service A and service B (i.e., $A \rightarrow B$) is represented by precedence edge 1. Generally, a precedence edge has a source service set and a destination service set. Each service in the source service set is the predecessor of each service in the destination service set. In other words, a precedence relation exits from a service in the source service set and leads to a service in the destination service set.

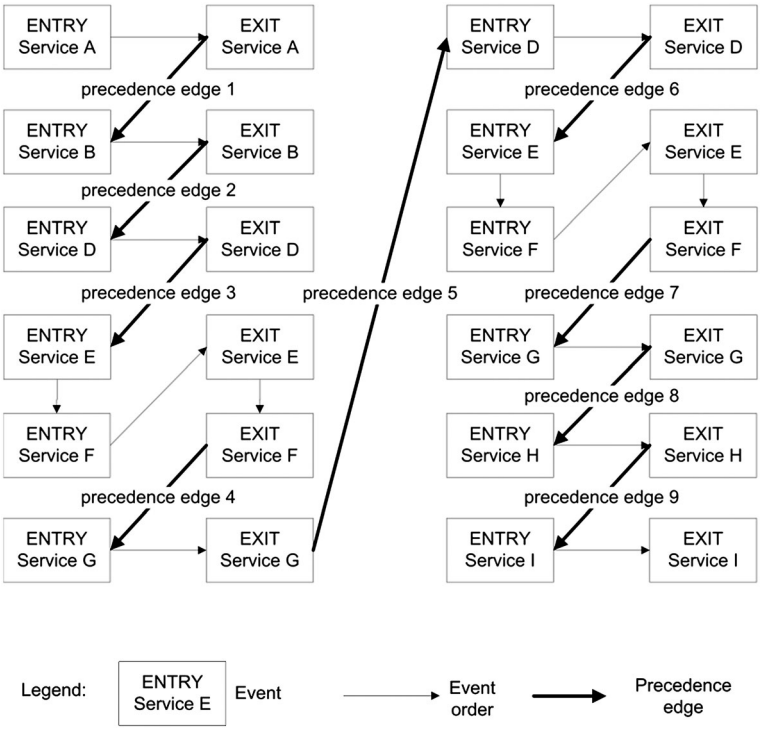


Figure 6. Identifying precedence and parallelism using event graph.

DOI: 10.1002/smr

To identify the source service set, we find all EXIT event nodes that can reach the precedence edges without going through an ENTRY event node. The services represented by such EXIT event nodes belong to the source service set. For the example shown in instances 3, 7, and 8 in Figure 5, precedence edge 4 has a source service set containing two services, namely services E and F. To identify the destination service set, we locate all ENTRY event nodes that can be reached from the precedence edge without going through an EXIT event node. The services represented by such ENTRY event nodes are included in the destination service set. For the example shown in Figure 5, precedence edge 4 has a destination service set containing service G.

In nonconcurrent service executions, the source service set and the destination service set only have one service. Thus, one precedence edge indicates exactly one precedence relation. In concurrent service executions, the source service set and destination service set may have more than one service. A precedence edge may indicate multiple precedence relations. For the example shown in Figure 4, from precedence edge 4, we can infer two precedence relations, namely $E \rightarrow G$ and $F \rightarrow G$.

Using the aforementioned approach, we construct an event graph for each execution instance shown in Figure 5. The identified precedence relations for each instance are shown using arrows in Table II.

4.1.2. Iterative Relation. A loop exists in the execution flow graph when several services are repeated in an execution instance. For example, the execution flow of execution instance 7 contains a loop because some services are repeated. An iterative relation is a special type of precedence relation as it leads to a successor service that is the entry service of a loop in the execution flow. An iterative relation is initially identified as a precedence relation using an event graph. For the example in Figure 6, the precedence edge 5 indicates a precedence relation $G \rightarrow D$, which is actually an iterative relation. To distinguish an iterative relation from a precedence relation, we use the depth-first traversal to traverse the execution flow starting from the first service in the execution instance. The traversal algorithm traverses the edges representing precedence (i.e., the edges with arrows in Table II). During the depth-first traversal, if an edge leads to the ancestors of the current node, it indicates the entry service of a loop. We identify such an edge as an iterative relation. For the example shown in Table II, the execution flow of execution instance 7 contains an iterative relation from service G to service D (i.e., $G \rightarrow D$).

4.1.3. Parallelism Relation. A parallelism relation represents the concurrent execution of two services. A parallelism relation is denoted by an undirected edge between services A and B (i.e., A–B) in the execution flow graph. A parallelism relation is also identified through a precedence edge in the event graph. When there are multiple services in the source service set, a parallelism relation exists between each pair of the services. Similarly, parallelism relations can be identified in the destination service set. For the example shown in Figure 7, we infer a parallelism relation, namely E–F from the two services E and F, in the source service set of precedence edge 4.

Using this approach, we identify parallelism relations in each execution instance shown in Figure 5. The results listed in Table II show that a parallelism relation is captured in the execution instances 3, 7, and 8 because the services are concurrently executed as shown in Figure 5. Services in parallel can be executed in any order. In the worst case, all the services are executed sequentially. Therefore, this technique may not detect all parallelism relations because some parallel services can execute in sequence. We address this by merging execution flows.

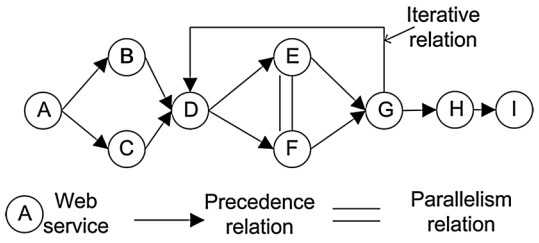


Figure 7. An example overall execution flow.

DOI: 10.1002/smr

4.2. Merging Execution Flows

To describe all possible execution flows among services in a service composition pattern, we merge execution flows of all execution instances that contain the pattern. The overall control flow contains all services in the pattern. For example, by merging all execution flows listed in Table II, the overall execution flow shown in Figure 7 contains all services. However, conflicts can occur during the merging process. We use the following rules to address the conflicts:

1. A pair of services has bidirectional precedence relations (e.g., an edge from service A to service B and an edge from service B to service A). This conflict occurs when the parallel services are misidentified as precedence relations because they are not concurrently executed. We replace such bidirectional precedence relations with a parallelism relation. In Table II, two precedence relations connect service E and service F (i.e., $E \rightarrow F$ and $F \rightarrow E$) after merging instances 2 and 4. We replace them with a parallel relation between services E and F because services E and F can be executed in a random order.
2. Two services are in both a precedence relation and a parallelism relation. The conflict occurs when the parallel services are misidentified as a precedence relation when they are not concurrently executed in some execution instances. The parallel relation covers the cases where the services are executed in sequence. We remove the precedence relation. For example, services E and F are detected as in parallel in instance 3. However, a precedence relation between service E and service F is identified in execution instance 1 because service E is executed before service F in this particular execution instance. The parallel relation prevails.

4.3. Inferring Control Flow Structures

To enable SOA developers to use the identified service composition patterns to compose new applications, we enhance the overall execution flows with control flow structures (i.e., sequential structure, loop structure, parallel structure, and alternative structure). We use the depth-first traversal algorithm to traverse all services in the overall execution flow by following edges representing precedence relations. The starting service is the service executed before any other services. The traversal begins at the starting service in the overall execution flow. During the traversal, we identify different types of control flow structures using edges representing precedence relations, iterative relations, and parallelism relations.

4.3.1. Sequential structure. When a service emanates an edge denoting a precedence relation, this service is in a sequential structure with its successor service. For the example shown in Figure 7, services H and I are in a sequential structure.

4.3.2. Loop structure. Services in a loop structure can be repeated in an execution instance. We identify a loop structure using the iterative relation detected in the previous subsection. An edge representing an iterative relation emanates from the exit service and leads to the entry service of a loop structure. For the example shown in Figure 7, the iterative relation between services G and D (i.e., $G \rightarrow D$) indicates that the services along the path from service D to service G are in a loop structure.

4.3.3. Parallel structure. In a parallel structure, multiple execution paths can be executed concurrently and invoked in any order. When two or more execution paths have common starting and ending services and parallelism relations exist between services in different paths, we identify these paths as in a parallel structure. For the example shown in Figure 7, the execution path along services D, E, G (i.e., $D \rightarrow E \rightarrow G$) and the execution path along services D, F, G (i.e., $D \rightarrow F \rightarrow G$) are converted to a parallel structure because services E and F are in a parallelism relation. Services E and F can be executed concurrently in any order, and both need to be completed before performing service G.

4.3.4. Alternative structure. Multiple possible execution paths are executed under a certain condition; that is, only one path is executed at one time. When multiple execution paths branch out from one service, the execution paths can be either in a parallel structure or in an alternative structure. If these

DOI: 10.1002/smr

paths are not identified as a parallel structure (i.e., no parallel relation exists between different paths), we identify it as an alternative structure. For the example shown in Figure 7, the path along services A, B, D (i.e., $A \rightarrow B \rightarrow D$) and the path along services A, C, D (i.e., $A \rightarrow C \rightarrow D$) are recognized as an alternative structure.

Using this approach, a control flow structures can be inferred from the overall execution flow. To ease the reuse of a service composition patterns in practice, we use Business Process Execution Language to represent the identified service composition patterns along with the control flow structures. For example, control flow structures are identified from the overall execution flow shown in Figure 7, and the resulting service composition pattern is depicted in Figure 8.

4.4. Inferring Structural and Functional Similarity between Recovered Patterns

We identify the similar patterns in two steps: (i) check a functional similarity between the services in two patterns; and (ii) find a structural similarity among two patterns.

4.4.1. Identifying functional similarity between operations in two patterns. To compare the functional similarity of two operations of Web services, we examine the types, messages, and ports [19] specified in the Web Services Description Language (WSDL) documents of Web services. Functional similarity between operations is defined as the similarity between the names of operations extracted from WSDL ports, the types of the input parameters, and the types of the output parameters extracted from WSDL types and WSDL messages. We also check the number of the input parameters and the output parameters.

The name of an operation in WSDL (e.g., "getWeatherInformation") is generally the concatenation of words that declare the functionality of the operation. The operation name in the service description may follow different conventions and is generally a compound word (e.g., "findCity"). The rules to decompose the names are given in Table III. After decomposing words, we use Porter stemmer [20], which is the process for reducing inflected (or derived) words to their stem, base, or root form. For example, the words "fishing", "fished", "fish", and "fisher" have the same root word, "fish". For the set of words used in the operations, we identify the semantic similarity between the words appearing in the names of operations using WordNet [21]. WordNet is a lexical database that groups words into a set of synonyms and connects words to each other via semantic relations:

1. Hypernym: describes a kind-of relation between words. For example, "red" and "color" have a hypernym relation because red is a color.

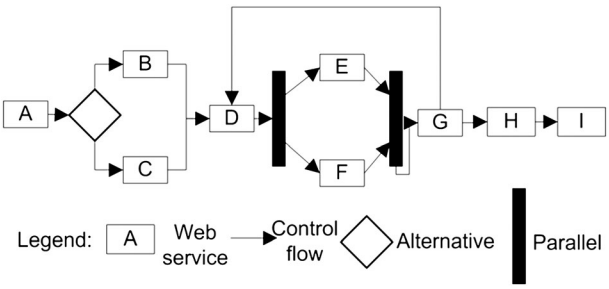


Figure 8. Identified service composition pattern with control flow structures.

Table III. Rules used to decompose the words.

Rule	Name	Word
CaseChange	FindCity	Find,city
CaseChange	getWeather	Get, Weather
Suffix containing number	City1	City
Underscore separator	Find_city	Find, City

DOI: 10.1002/smr

2. Hyponym: describes a type-of relation between words. For example, "vehicle" is a hyponym of "car".
3. Holonym: describes a whole-part (i.e., partOf) relation between words. For example, "tree" is a holonym of "bark".
4. Meronym: describes a part-whole relation between words. Meronym is the inverse of holonym. For example, "finger" is a meronym of "hand" because a finger is a part of a hand.

If the words have hypernym and hyponym relations, we consider that they are the same. The similarity of the names of two operations is evaluated as given by equation (3). It measures the ratio between the number of common words in the operation names and the total number of words in both operation names. The value of operational similarity (i.e., OS) is 1 in case the names of the operations have the exact match and 0 when there is no match.

$$OS(op1, op2) = \frac{|words(op2) \cap words(op2)|}{|words(op1) \cup words(op2)|} \tag{3}$$

OS refers to the operational similarity; *op*1 and *op*2 represent operation names; *words* (*op*1) and *words* (*op*2) are the set of words appearing in the operation names *op*1 and *op*2.

$$SI(op1, op2) = \frac{|InputParameters(op1) \cup InputParamters(op2)|}{|InputParameter(op1) \cup InputParameters(op2)|} \tag{4}$$

SI refers to the similarity in input parameters; *op*1 and *op*2 are the operations; *InputParameters* (*op*1) and *InputParameters* (*op*2) refer to the names of input parameters in operations *op*1 and *op*2.

$$SO(op1, op2) = \frac{|OutputParameters(op1) \cup OutputParamters(op2)|}{|OutputParameter(op1) \cup OutputParameters(op2)|} \tag{5}$$

SO refers to the similarity in output parameters; *op*1 and *op*2 are the operations; *OutputParameters* (*op*1) and *OutputParameters* (*op*2) refer to the names of output parameters in operations *op*1 and *op*2.

Functional similarity is also a measure of the type, the number of input parameters and output parameters between the operations. Equation (4) gives the similarity measure between the input parameters of two operations. It is the ratio of the number of the same input parameters to the total number of the input parameters of two operations. The value of *SI* is 1 in case that the names of the operations have the exact match and 0 when there is no match. Similarly, the similarity between the output parameters is given in equation (5). *SO* is the ratio of the common types of the output parameters between the operations by the total number of output parameters in the operations. The value SO is 1 when the names of the operations have the exact match and 0 when there is no match.

$$\text{FunctionalSimilarity}(op1, op2) = \frac{OS(op1, op2) + SI(op1, op2) + SO(op1, op2)}{3} \tag{6}$$

OS refers to operation similarity between operations *op*1 and *op*2; SI refers to the similarity in the input parameters of the operations *op*1 and *op*2; *SO* refers to the similarity in the output parameters of the operations *op*1 and *op*2.

The overall functional similarity between the two operations is specified in equation (6). The values for operational similarity (*OS*), input parameter similarity (*SI*), and output parameter similarity (*SO*) fall within the range of 0 and 1. We normalize functional similarity shown in equation (6) by dividing it by 3. Therefore, the result of the functional similarity is between 0 and 1. 1 indicates exact match between the two operations and 0 shows no match.

4.4.2. Identifying structural similarity between the recovered patterns. For all the recovered patterns within a domain, we identify structurally similar patterns. We are interested in comparing the structure of the control flow graphs of the patterns. In a control flow graph, a node represents a service or a control flow structure (i.e., alternative, parallel, and iterative nodes); and an edge denotes a control

DOI: 10.1002/smr

flow that links two services or connects a control flow structure and a service. Two graphs may have three kinds of structural similarity:

1. Exact match: the control flows match perfectly between the patterns. An example is shown in Figure 9(a). The two control flow graphs have the exactly same structure.
2. Plug-in match: one graph matches a part of the other graph. For the example as shown in Figure 9 (b), the nodes inside the dotted area have the same control flow.
3. Subset match: two graphs have a subset of common control flows. For the example as shown in Figure 9(c), both graphs share some common control flows enclosed by the dotted area.

In the case of exact and plug-in matches, a complete pattern (e.g., the one with the smaller number of nodes) can be extracted from both patterns. For the example shown in Figure 9(b), the left side graph can be extracted from the right side graph. The extracted pattern can be treated as more generalize patterns between the two patterns if the functionality of the common services is the same. In the case of subset match, no complete overlapping between the patterns can be found. Therefore, we consider patterns of subset matches as different patterns. For the exact and plug-in matches between two graphs, we use the depth-first graph traversal algorithm to visit the nodes in both patterns to compare the structure of both control flow graphs and examine the functionality of the corresponding services with the similar control flow structure. The traversal terminates when all the nodes are visited in both graphs.

Figure 10 shows the algorithm to find a generalized pattern from two control flow graphs of service composition patterns. We count the number of the nodes in the control flow graphs of two patterns. The control flow graph that contains the smaller number of nodes can be matched with any part of the control flow graph that has the larger number of nodes. We iteratively select the control flow graph with the minimum number of nodes to compare with a larger control flow graph until we compare all the control flow graphs. We identify the plug-in match between the control flow graphs of different sizes. If the two control flow graphs contain the equal number of nodes, then we search for the exact match between the graphs. The following steps show the process of comparing structural similarity between two patterns:

1. *Select starting points for the graph traversal*. We select a starting point for the traversal by counting the in-flow of the nodes. The nodes with no predecessors are selected as the candidates for the starting nodes for the traversal. When more than one candidate has no predecessors, we randomly select one. We compare every node in the larger graph with every node in the smaller graph. For the example shown in Figure 11, graph G1 is the smaller graph, and graph G2 is the larger graph. Node A in G1 and node M in G2 are selected as the starting nodes. We traverse graph G1 starting from node A and graph G2 from node M.
2. *Compare the functionality of two corresponding nodes in both graphs*. There are two kinds of nodes in the graphs: (i) control flow structures and (ii) service nodes. If a node is a control flow structure, we treat the same type of control flow constructs (e.g., loop, alternative, and parallel) as the nodes delivering the same functionality. For a service node, we further measure the functional similarity between them as discussed in the previous subsection. If the functional similarity

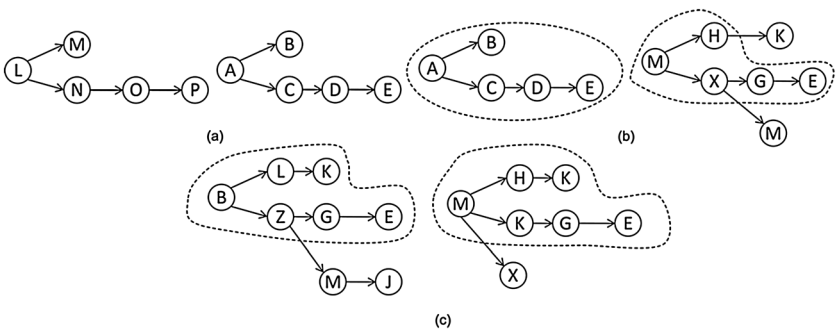


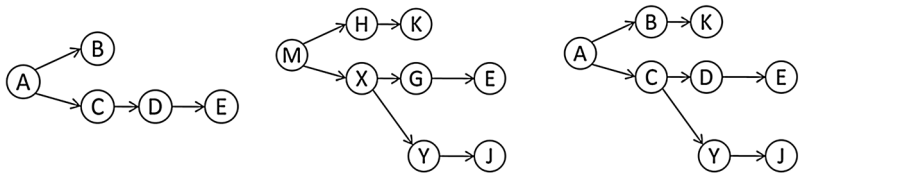
Figure 9. Different types of structural similarity.

DOI: 10.1002/smr

```
Function: findStructuralSimilarity
Input:    Spattern → A control flow of service composition pattern with least number of nodes
          Epattern →  A control flow of service composition pattern with maximum number of nodes
Output:   Exact/Plugin in control flow in SPattern and EPattern

Algorithm
1.  List Nodes=DFTraversal(SPattern)    //Depth First traversal of SPattern
2.  eStart=getStartingNodes(EPatern)  //Starting point to traverse a graph EPattern
3.  Stack openlist={eStart}
4.  while (EPattern has more nodes to visit)
5.          while (openlist.top is visited)
6.                  openlist.pop();
7.          End while
8.          vVertex=openlist.pop(); //visited vertex in EPattern
9.          For each non-visited node with an edge from vVertex
10.                 openlist.push(vertex);
11.         End For
            //checks if the list contains functionally similar nodes
12.         if(Nodes.contains(vertex))
13.                 EPattern.changeLabel(vertex.SPattern)
                    //Change vertex with functionally similar vertex from graph SPattern
14.         End If
15. End while
16. EPattern_adj= adjacencyList(EPattern)     //EPattern_adj is a adjacency list of EPattern
17. SPattern_adj= adjacencyList(SPattern)     //SPattern_adj is a adjacency list of SPattern
18. if(SPattern_adj ∩ EPattern_adj=SPattern_adj)
19.         return SPattern     //SPattern and EPattern have common control Flow
20. return -1                   //SPattern and EPattern are different
```

Figure 10. Algorithm to find the structurally similar patterns.



(a) Control flow graph of patterns (G1)

(b) Control flow graph of patterns (G2)

(c) Control flow graph of pattern G2 after renaming functionally similar nodes with G1 (G3)

Figure 11. Example of control flow graphs with the structural similarity.

between the two nodes is greater than a predefined threshold value (e.g., 75%), we consider that the two services deliver the same functionality. We rename the node in the larger graph with the name of the node in the smaller graph to make that services with the same functionality have consistent name across all the patterns. For the example shown in Figure 11, assuming that node A and node M have a similar functionality, we rename node M with A. The updated G2 is illustrated in a new graph G3.

3. *Expand the path length of functionally similar nodes in both graphs*. We expand the path length of the nodes with the same functionality by 1 to identify the successors of the nodes. We further compare the functional similarity among the successors as described in Step 2. For the example shown in G1, the successors of node A in graph G1 are nodes C and B. Similarly, the successors of node M in graph G2 are nodes H and X. We assume that nodes B and H are functionally similar and that nodes C and X are functionally similar. We rename node H in graph G2 with node B in graph G1 and replace node X in graph G2 with node C in graph G1.
4. *Recursively traverses the unvisited nodes*. We recursively traverse all the nodes along a path from the starting node until the functional similarity of the corresponding nodes in both graphs are lower than the threshold value or reach the end of the path in a graph.

DOI: 10.1002/smr

5. *Compute the adjacency list of both graphs when the traversal is complete*. An adjacency list is the representation of all edges in a graph as a list. Every entry in the adjacency list describes two nodes for an edge: one denoting the source node and the other denoting the destination node of the corresponding edge. For example, the adjacency lists for graphs G1 and G3 are shown as follows:

$$G1_{adj} = \{(A,B),(A,C),(C,D),(D,E)\} \text{is the adjacency list of graph G1, and } G3_{adj} = \{(A,B),(A,C),(B,K),(C,D),(C,Y),(D,E),(Y,L)\} \text{is the adjacency list of graph G3.}$$

6. *Compare the adjacency lists of both graphs*: If the adjacency list of the larger graph contains the adjacency list of the smaller graph, we choose the smaller graph as a generalized pattern. If the adjacency lists do not match, then both the control flow graphs are treated as different graphs. For example, graph G3 represents a new graph after we rename the functionally equivalent nodes with the same labels as the ones in graph G1. If $G1_{adj} \sqcap G3_{adj} = G1_{adj}$, then graphs G1 and G2 have the common control flows with the functionally similar services. Otherwise, there are no common control flows. Because the intersection of $G1_{adj}$ and $G3_{adj}$ is $G1_{adj}$, the services of similar functionality in graphs G1 and G2 can be generalized.

5. CASE STUDY

To demonstrate the effectiveness of our approach, we conduct a case study to (i) verify if our approach can accurately recover the control flow of a service-oriented application from execution logs, (ii) validate if our approach can accurately identify service composition patterns (iii) verify and validate if our approach can accurately find the patterns with functional and structural similarity, and (iv) conduct the performance evaluation of our approach.

5.1. Setup

In the case study, 93 service-oriented applications are studied. The applications are composed from 93 business processes provided by a business process management architect from our industrial collaborator [22]. An overview of these service-oriented applications is listed in Table IV. The applications specify activities used in various domains. The number of the applications in each domain is also listed in Table IV. For example, the financial management applications manage financial policies, procedures, account, and asset.

The execution logs are collected from the testing environment when testing the 93 Web services applications. The execution engine uses WebSphere Process Server installation. IBM WebSphere Process Server [14] is used to host the applications. In the process of testing, SoapUI [23], a standard SOAP-based invocation tool, is used to automatically execute the service-oriented

Table IV. Business processes used in case study.

Industrial domains	No applications	Description
Financial management	10	Manage financial policies, procedures, account, and asset
Human Capital Management	11	Manage recruiting new employees and administering existing employees
Supply chain management	10	Manage product, market research, and procurement of materials and services
Banking	11	Open bank accounts
Insurance	10	Handle customer's applications for insurance
Government	12	Administer employer/unemployment services
Retail	10	Manage inventory, product, order, and prices
Medical	9	Handle patient registration, triage and evaluation process, including subprocesses for blood pressure evaluation and blood tests
Telecommunication	10	Manage service provisioning process, including customer prequalification, credit check, and release of the order

DOI: 10.1002/smr

applications for a large number of times. We register the events for each task specified in the business processes through the Common Event Infrastructure (CEI) built in the IBM WebSphere Process Server. The CEI provides consistent, unified APIs and infrastructure for creating, transmitting, and distributing a wide range of events. The registration of events is conducted in the business processes without accessing the source code of the service that implements the task. The CEI can record the triggered events and the parameter passing between the service invocations in the logs. The execution logs are collected in a default directory in the process server.

In our case study, two of the co-authors inspect the results. One evaluator examines the service composition patterns recovered from the execution logs. The other evaluator checks the results from identifying structurally and functionally similar patterns. Our evaluators have 2 years of experience in developing business applications and studied the business processes of the domains used in our case study.

5.2. *Results of Recovering Control Flows*

We recover the control flows of each service-oriented application from the execution logs that contain 10,000 execution instances for each application. The accuracy of the recovered control flows is manually verified by comparing with the original business processes. Specifically, one of the co-authors manually check the recovered control flows of each application and compared them with the control flows of the original business processes used to compose the application. Our approach can correctly handle most special cases, such as self-loop (i.e., loop containing only one service). As a result, our approach accurately identifies the control flows of 92 out of 93 service-oriented applications listed in Table IV. Table V shows the number of nodes, edges, parallel relations, and alternative relations in the identified control flows. Each service has two events: ENTRY and EXIT events. The number of events in the execution logs is proportional to the number of services in the execution applications. However, our approach fails to handle a special case in which an alternative structure has two branches with identical services in different orders. Specifically, in the first branch, service A was executed before service B. In the other branch, service B was executed before service A. As a consequence, services A and B appear in the execution log in a random order. Our approach misidentifies services A and B as in a parallel structure because we use the random execution order as one of the criteria to identify the parallel structure.

5.3. *Results of Identifying Service Composition Pattern*

To verify if our approach can correctly identify service composition patterns from the 93 service-oriented applications, we use a testing environment with varying numbers of execution instances for the applications and apply our approach. We measure the correctness of the identified pattern service sets and the correctness of the identified pattern control flows.

5.3.1. Correctness of the identified pattern service sets. We use three metrics to measure the correctness of the identified pattern service sets, that is, precision, recall, and coverage.

Precision is defined in equation (7). It measures whether our approach can correctly identify the execution instances that contribute to a pattern service set. The precision should be high if our

Table V. Summary of identified control flows.

No	Domain of patterns	No applications	No nodes	No edges	No parallel relations	No alternative relations
1	Financial management	10	120	170	20	30
2	Human Capital Management	11	115	153	20	19
3	Supply chain management	10	152	195	20	28
4	Banking	11	158	220	33	20
5	Insurance	10	129	187	22	22
6	Government	12	118	160	22	25
7	Retail	10	140	140	19	25
8	Medical	9	161	203	38	30
9	Telecommunication	10	130	183	20	21

DOI: 10.1002/smr

approach is correct. Equation (8) gives the recall metric that is the fraction of relevant instances retrieved using our approach. Recall is a measure of completeness of our approach.

$$\text{precision} = \frac{\textit{correctly identified instances}}{\textit{identified execution instances}} \tag{7}$$

$$\text{recall} = \frac{\textit{correctly identified instances}}{\textit{total execution instances}} \tag{8}$$

Coverage is defined in equation (9). It calculates whether the identified pattern is widely adopted among the applications. To judge the correctness of our approach, the coverage is used in conjunction with knowledge about the functionality of the applications which generates the logs. If we know that the execution logs are collected from a set of applications that fulfill similar functionalities, we could expect the coverage to be high. In contrast, if the execution logs are collected from a set of applications with diverse functionalities, we expect the coverage to be low.

$$\text{coverage} = \frac{\textit{instances that contribute to a pattern}}{\textit{total execution instances}} \tag{9}$$

5.3.2. Correctness of the identified pattern control flows. We use accuracy to measure the correctness of the control flow of an identified pattern. It is defined as equation (10). Among all the execution instances that contribute to a pattern, we count the correct execution instances. An execution instance is correct if the control flow of its underlying application is a super graph of the control flow of the identified pattern.

$$\text{accuracy} = \frac{\textit{correct execution instances}}{\textit{execution instances that contribute to a pattern}} \tag{10}$$

The results are shown in Table VI. The high precision shows that our approach can accurately identify pattern service set. The moderate recall shows that our approach misses patterns in some of the execution instances. Figure 12 shows the recall and precision graph in nine domains. The coverage of a pattern service set is moderate because in our case study, the execution log is collected from a combination of similar and different applications. Therefore, the patterns are not widely used in all applications. The accuracy of a pattern control flow is high but not 100% accurate

Table VI. Results of identified service composition patterns.

No	Domain of patterns	No total execution instances (K)	No execution instances related to a pattern	Accuracy of a pattern service set			Accuracy of a control flow (%)	% reuse
				% precision	% recall	% coverage		
1	Financial management	15	10,285	100	75	68	91	40
2	Human Capital Management	16	11,110	100	68	69	87	54
3	Supply chain management	15	9778	100	63	65	85	50
4	Banking	16	10,413	100	71	65	92	54
5	Insurance	15	9559	100	68	63	88	60
6	Government	17	12,423	100	72	73	90	25
7	Retail	15	10,230	100	70	68	92	60
8	Medical	11	7089	100	74	64	91	55
9	Telecommunication	12	8011	100	72	66	87	40

DOI: 10.1002/smr

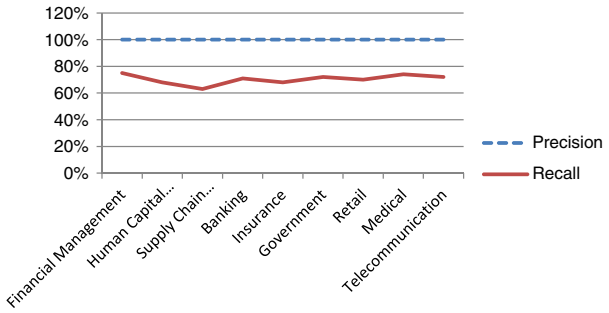


Figure 12. Precision and recall results for identifying service composition patterns.

because of cases where two or more services are arranged slightly differently between similar applications, although they share the pattern service set. Suppose that two services (i.e., A and B) are in a parallel structure in some applications. However, in other applications, A and B are arranged as a sequence (i.e., $A \rightarrow B$). We cannot detect the conflicting situation and treat the services in a sequential relation as a parallel relation. However, a sequential relation is essentially a special case in the parallel structure of the two services. The percentage of reuse shows number of times that a pattern occurred in a domain. The percentage of reuse varies between 25% to 60%.

5.4. *Results of Structural and Functional Similarity among the Recovered Patterns*

Our objective is to find functional and structural similarity among the recovered patterns. We measure the precision and recall to evaluate the performance of our approach. Precision is a ratio of the number of patterns being correctly identified by our approach with respect to the total number of functionally and structurally similar patterns identified by our approach (i.e., retrieved patterns). The definition of precision is given in equation (11). Recall evaluates a number of patterns correctly identified by our approach with respect to the total number of functionally and structurally similar patterns (i.e., relevant patterns). The definition of recall is shown in equation (12). To verify if our approach can correctly identify the structural and functional similarity, we manually analyzed all the recovered patterns and categorized them according to the structural and functional similarity. Then we compared our manually classified results with the results obtained from our approach.

$$pattern_precision = \frac{\left|\{relevent\ patterns\} \cap \{retrieved\ patterns\}\right|}{\left|(retrieved\ patterns\}\right|} \tag{11}$$

$$\text{pattern}_{\text{recall}} = \frac{\left|\{relevant\ patterns\} \cap \{retrieved\ patterns\}\right|}{\left|\{relevant\ patterns\}\right|} \tag{12}$$

Table VII shows the results of precision and recall for our approach to find structurally and functionally similar patterns among the recovered patterns. We randomly selected a small subset (10%) of patterns and calculated the functional similarity with different threshold values starting from 15% to 95% percentage. We increase the threshold by 10% each time. We found that the optimal results are at the thresholds of 75% and 85%. Hence, both 75% and 85% are used as two threshold values for measuring the functional similarity between two services. As shown in Table VII, the 75% threshold value can produce higher precision and recall. Our approach can find the structurally and functionally similar patterns with a high precision (i.e., an average, 88% precision with the threshold value of functional similarity set to 75%). The main reason for not gaining 100% precision is caused by the errors in the functional similarity comparison.

The functional similarity is based on the similarity between the words present in the operation name, input parameters, and output parameters. However, some of the words used in the operation names may

DOI: 10.1002/smr

Table VII. Precision and recall of identifying functionally and structurally similar service composition patterns.

No	Domain of patterns	Accuracy of functional and structural similarity between patterns (Functionality similarity threshold = 75%)		Accuracy of functional and structural similarity between patterns (Functionality similarity threshold = 85%)	
		% precision	% recall	% precision	% recall
1	Financial management	90	69	87	54
2	Human Capital Management	83	71	85	43
3	Supply chain management	84	65	83	29
4	Banking	88	67	85	50
5	Insurance	90	69	77	54
6	Government	81	69	88	62
7	Retail	91	73	85	40
8	Medical	92	75	90	56
9	Telecommunication	100	76	100	59
	Average	88	71	86	50

have multiple meanings. For example, the operation names "*buyBook*" and "*bookTickets*" contain a common word, "*book*". However, word "*book*" is used in different contexts. The word "*book*" in "*buyBook*" is used as a noun, and the "book" in "*bookTickets*" is a verb. Our approach detects that both phrases share a word and hence are functionally close. Similarly, the same numbers and types of input parameters and output parameters can increase the functional similarity of two unrelated operations.

The average recall of our approach is 71% when the threshold value of functionality similarity is set to 75%. Our approach could not identify the functionally similarity between two operations if the words appearing in the operation names are not available in WordNet. For example, an operation name may contain abbreviations and terms (e.g., *DHL, UPS*). Such words are not found in WordNet. Moreover, the use of ambiguous words causes difficulties to identify functional similarity among operations. For example, regional words, such as *livre* that means *book* in French, are not recognized in WordNet. The operations with functional similarity may have different input parameters and output parameters. Our approach is unable to classify such operations correctly. Figure 13 shows the graph of the precision and recall for the functional and structural similarity between patterns for threshold 75% and 85%. APPENDIX A gives a summary of the service composition patterns identified by our approach. We list the domain of application, its diagram, a short description, and the application of each pattern.

5.5. *Performance of our Approach for Identifying Patterns*

In this case study, we evaluate the performance of our graph-based approaches to identify the patterns from the execution logs. The case study is conducted on a laptop with Intel Core 2 Duo CPU of

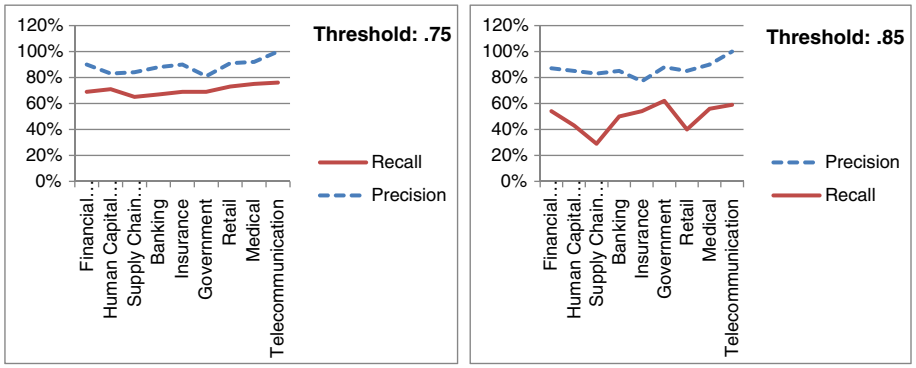


Figure 13. Precision and recall results for functional similarity between patterns with different thresholds.

DOI: 10.1002/smr

2.66 GHz and 4 GB RAM. We sample the execution instances with varying sizes from 1 to 10 K. We identify the control flows in the execution instances. The number of nodes in the recovered control flows ranges from 4 to 20. The control flows are then evaluated to identify the patterns. Table VIII shows the time required to identify the patterns from the varying size of the execution logs. As shown in Table VIII, the time needed to identify the patterns is dependent on the number of control flows. The cost to identify the patterns is expensive when running on such a hardware configuration. In the future, we will optimize our graph-based approaches to reduce the cost of identifying patterns from execution logs.

5.6. Threats to Validity

In this section, we discuss the limitations of our approaches and the threats that may affect the validity of the results of our case study.

In our case study, two of the co-authors inspect the results. Our evaluators have experience in developing business applications and studied the business processes of the domains used in our case study. It is difficult to find a professional analyst to perform such a time-consuming task. In the future, we plan to invite business analysts to conduct a smaller case study and provide feedback on the recovered service composition patterns. The manual verification introduces bias because a single evaluator could make mistakes. Although there were two evaluators in our case study, each evaluates different results for the case study. Therefore, our evaluation is based on a single person. We should have recruited additional people for the evaluation. Unfortunately, we were not able to recruit more evaluators with sufficient knowledge about service-oriented applications and who can spend considerable time to manually inspect our results.

To generalize our results to business processes in other domains, we chose to study systems with a variety of domains to help ensure the generality of our results. Our approach is dependent on the granularity of the generated logs from the business process execution engine. We only used two threshold values (i.e., 75% and 85%) to assess the precision and recall of our approach for identifying the functional and structural similarity among the service composition patterns. We manually examine a sample set of WSDL documents to evaluate the functional similarity between operations and find that 75% and 85% can give the optimal results in the precision and recall. In the future, we plan to evaluate other possible threshold values with a larger sample size.

6. RELATED WORK

A great amount of effort has been devoted to identifying service composition patterns. In this section, we discuss the related work in four areas: (i) mining service composition logs; (ii) identifying similarity among service composition patterns; (iii) model-based service composition patterns; and (iv) analysis of execution traces of software components.

6.0.1. Mining service composition logs. In the business process management domain [24, 25], a composition pattern catalogs different forms of service interactions through control flows (e.g., sequential pattern and loop pattern). Such a pattern does not refer to a particular existing service. In our work, we identify service composition patterns that consist of operations from Web services and

Table VIII. Size of the execution logs versus time taken to identify patterns.

No execution instances (K)	No control flows	Time taken to identify patterns (min)
1	4	.54
2	5	1.21
3	5	1.58
4	6	2.3
5	6	3.4
10	7	9.33

DOI: 10.1002/smr

the control flows among the operations. Hence, the identified patterns are more concrete and ready to use for composing or improve applications.

Liang *et al*. [26] mine service association rules from service transactional data. Their approach aims to discover a binary association relation between two service sets. Agrawal *et al*. [5] propose a technique for mining business processes from execution logs. This approach builds a dependency graph using the order in which activities are recorded in the historical log. For example, activity A depends on another activity B if A can only happen after B. However, this work did not further identify control flow structures based on the dependency graph. Cook and Wolf [6] develop a similar approach to recover business processes from event logs. The approach is capable of identifying parallel structures assuming that business tasks in a parallel structure are invoked in random order. Schimm [7] recovers hierarchically structured business processes. Aalst *et al*. [9] propose an approach to recover business processes using Petri net theory. Similar to Agrawal [5], Aalst *et al*. use tasks in the business processes logs to construct Petri net to indicate the relationship among tasks. Aalst *et al*. can identify dependency relations, nonparallel relations, and parallel relations to describe the execution order among tasks.

In a heuristic approach [11], metrics are used to construct business processes using logs. Specifically, a dependency/frequency table is constructed on the basis of the order of occurrences between each pair of tasks. The graph representing task execution order is induced from the dependency/frequency table. This approach can handle execution data with noise. Greco *et al*. [27] mine multiple variants of processes to represent different usage scenarios. Bose *et al*. [28] propose an approach to cluster instances to handle less structured process model. Francescomarino *et al*. [29] develop an approach to recover business processes of Web applications by mining GUI-form traces.

Similar to existing approaches, our approach can identify sequential, alternative, parallel, and loop control flow structures. However, our approach is different from the existing approaches in the following aspects. (i) We improve the technique for identifying parallelism. Existing approaches detect parallelism relations between services if the services appear in random orders in different execution logs [6, 9]. Such approaches require a large number of execution logs reflecting all possible execution orders among services. Furthermore, not all runtime execution environments support the execution of parallel services in random order. Our approach uses the ENTRY and EXIT service invocation events to detect concurrency. In the cases where paralleled services are executed concurrently, our approach can identify parallelism using a single execution of a service-oriented application without the assumption of the random execution order among parallel services. (ii) Different from the existing approaches that focus on recovering a single entire process, we mine the frequently executed patterns used in multiple service-oriented applications. Similar to the existing approaches [5, 9], we use the techniques to recover control flow of the pattern.

6.0.2. Identifying the similarity among service composition patterns. Techniques, such as text document matching, schema matching, and software component matching are used to identify functional similarity among operations of Web services. Several methods [30, 31] are developed to capture clues about the semantics of schemas and suggest schema matches using linguistic analysis, structural analysis, and domain knowledge. Our work focuses on identifying the functional similarity among the operations of Web services.

There are various methods to identify the semantic similarity between the Web service interfaces [32, 33, 30, 34–36]. Natallia [30] provides a detailed survey on different approaches to compute similarity between Web service interfaces. Nayak [33] proposes a method to calculate the similarity between Web services using the Jaccard coefficient that measures the similarity between sample sets. The Jaccard coefficient is defined as the size of the intersection divided by the size of the union of the sample sets. Dong *et al*. [32] purpose a method to cluster operations using the parameters of the operations. The parameters tend to reflect the same concept if they often occur together. Liu *et al*. [34] apply text mining techniques to extract features from service description files to identify similarity among Web services. Different from aforementioned approaches, our approach identifies the similarity among operations of Web services by examining the similarity in the operation name, inputs parameters, and output parameters.

DOI: 10.1002/smr

Dijkman *et al*. [37] investigates two approaches for business process alignment. Their approaches are based on lexical matching and error correcting graph matching. Our matching approach first identifies the structurally similar business processes from the execution logs and then uses semantic analysis of those business processes to find functionally similar services. Nejati *et al*. [38] propose a heuristic based on both static and behavioral properties to find the matching state charts. Minor *et al*. [39] propose the case-based approach to represent an index-based retrieval of past business processes to give authoring support for adaptation of recent business processes instances. Our approach is based on the examining the execution logs of business processes. Similar to our approach, Juan [40] propose the Business Process Execution Language process matchmaking by using graph matching techniques. Brockmans *et al*. [41] provide the representation of Petri nets in Web Ontology Language to semantically enrich the business process models. They propose a technique to semantically align business processes to semi-automated interconnectivity of business processes.

6.0.3. Model-based service composition patterns. Research efforts have been devoted to describing business processes patterns in different formalisms. Model checking techniques are used to compare the execution of two state machines. Schäfera *et al*. [42] propose a tool HUGO to automatically verify whether the interactions expressed by collaboration can be realized by a set of state machines. HUGO uses the model checker SPIN to verify the model against the automata. Puhlmann *et al*. [43] discuss the application of the π-calculus for describing the behavioral perspective of business processes patterns. The formalizations are then used as a foundation for pattern-based business processes execution, reasoning, and simulation. Stefansen *et al*. [44] present a representation of the patterns in CSS (a subset of π-calculus). In general, process calculi are used for specifying business processes, then model checking for their verification, and automata for scheduling. Different from the model-based approaches, we use graph-based and linguistic-based approaches to find the patterns in the execution logs.

A conversation is the global sequence of messages exchanged among the components of a distributed system. Conversations provide an intuitive and simple model for analyzing interactions among composite Web services. Bultan *et al*. [45] use a model-based approach to identify the interaction among Web services. Bultan *et al*. [45] use synchronizability and realizability analyses as two techniques for analyzing conversations. Synchronizability analysis is used to identify bottom-up Web service specifications for which asynchronous communication does not change the conversation behavior. Realizability analysis is used to identify top-down Web service specifications that are realizable using asynchronous communication. Bultan *et al*. [46] discuss that the approach of top-down and bottom-up conversation specifications can be automatically verified. Different from the conversation model, our approach identify service composition patterns that consist of operations from Web services and the control flows among the operations.

6.0.4. Analysis of execution traces. The analysis of execution logs [47–49] can be useful in many software engineering activities including debugging, feature enhancement, performance analysis, and any other tasks that help to understand system behaviors. We analyze the execution logs to recover the control flows of SOA applications. A rich body of work uses execution logs to model and extract designs of the software components.

Di Lucca *et al*. [48] present a reverse engineering approach to recover a use case model from object-oriented code. The approach identifies use cases by analyzing class method activation sequences triggered by input events and terminated by output events. Di Lucca *et al*. [47] propose an approach to abstract use case diagrams from execution logs of a Web application. Di Lucca *et al*. record Web Application executions in transition graph. The transition graph is analyzed along user sessions to cluster of navigated pages. Their main purpose for their work is to find the evolution of Web application. Hamou-Lhadj *et al*. [49] purpose an algorithm to detect utility components based on component dependency graph. Hamou-Lhadj *et al*. built behavioral design models from execution logs. Hamou-Lhadj *et al*. use case map is used represent behavioral model. We use execution logs to identify patterns in SOA application within a domain. The identified patterns are represented in Business Process Modeling Language (BPML) notation.

DOI: 10.1002/smr

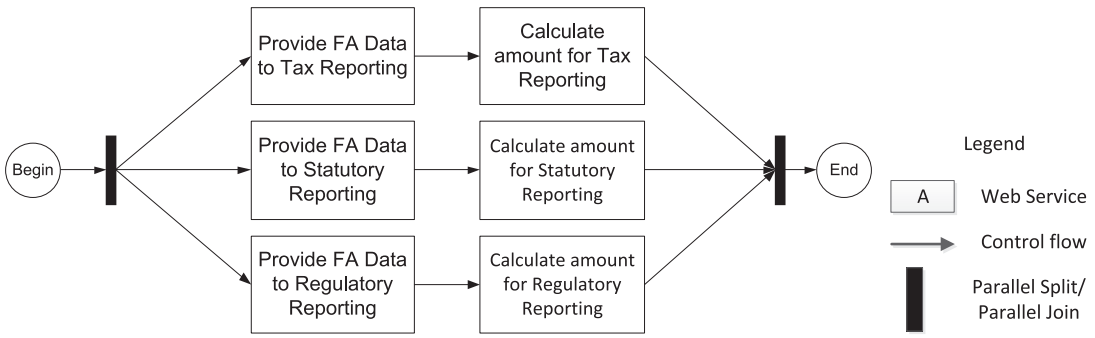
7. CONCLUSION AND FUTURE WORK

In this paper, we present an approach for identifying service composition patterns from execution logs. Service composition patterns record the best practices in designing and developing reliable service-oriented applications. To identify a pattern, we find a set of services frequently executed together along with their control flows. We also propose an approach to identify functionally similar patterns to represent the recovered service composition patterns independent from the actual services. Our approach helps service providers to identify the service composition patterns that are proven to be successfully applied in practice for developing new applications or refining their existing applications. The identified patterns facilitate the documentation and reuse of service composition patterns to improve the productivity of SOA developers. The case study demonstrates that our approach can effectively detect service composition patterns with accurate control flows and frequently used sets of services. The case study also shows that our approach can identify the functionally similar patterns precisely.

In the future, we plan to enhance our approach for identifying functional similarity by comparing conversations between the services. We will also enhance the process of identifying patterns from the execution instances using the data flows between the services. For example, by considering data flow relations among services, we could capture the roles of the tasks in a pattern, such as wrapper, dispatcher, and fault tolerance. We would like to investigate the possibility to get the patterns directly from the message exchange between services, in addition to analyzing of the control flows among the services. We also plan to facilitate the reuse of the identified patterns by enhancing the documentation of the identified service composition patterns. We plan to capture more detailed information for each pattern, such as the contexts and consequences of using the identified patterns.

APPENDIX A: Summary of the composition patterns identified

(1) *Name*: Financial report
Application domain: Financial management
Description: A pattern provides fixed asset data for tax, regulatory, or statutory purposes.
The business process of the pattern:

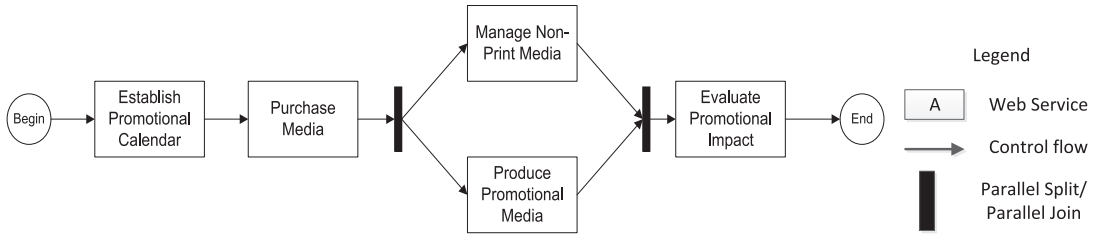


Application of the pattern: The pattern is used in the process that involves enforcing financial policies and procedures, asset management, and general accounting activities.

(2) *Name*: Product promotion
Application domain: Retail
Description: A pattern describes the steps for product promotion in a retail store.

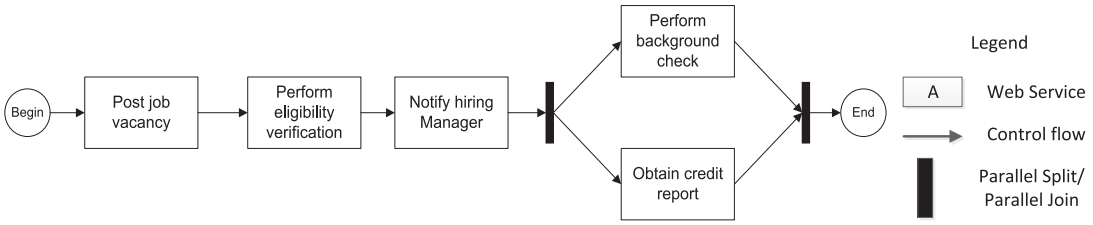
DOI: 10.1002/smr

The business process of the pattern:



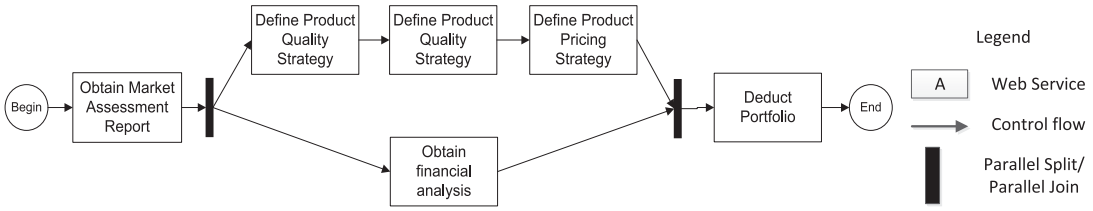
Application of the pattern: The pattern is used in the supplier forecasting process that generates supplier request, response review, and purchase order.

(3) *Name*: Recruitment
Application domain: Human Capital Management
Description: A pattern describes the steps for handling staffing process.
The business process of the pattern:



Application of pattern: The pattern is used by most of the recruitment process. The pattern is used for internal recruitment, external recruitment, and promotion.

(4) *Name*: Portfolio management
Application domain: Supply chain management
Description: A pattern conducts the portfolio management of products and services offered by an organization to its customers.
The business process of the pattern:

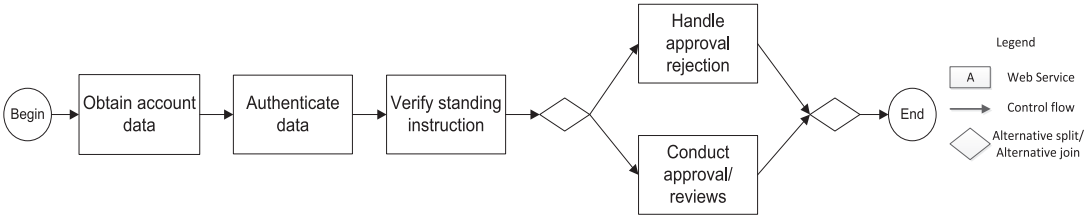


Application of the pattern: The pattern is used in the process involving market research, procurement of materials and services, contract negotiation, and supply management.

(5) *Name*: Loan processing
Application domain: Banking
Description: A pattern processes loan applications.

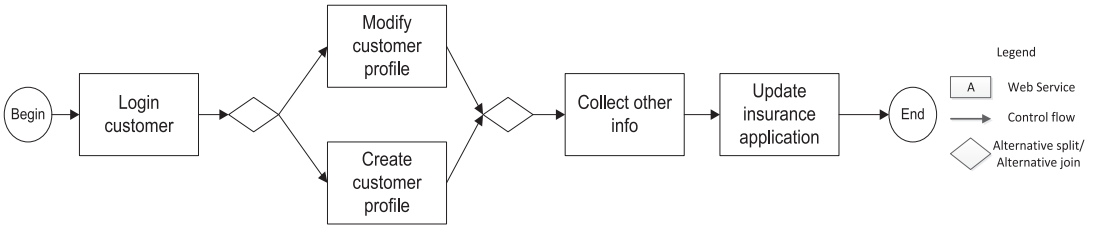
DOI: 10.1002/smr

The business process of the pattern:



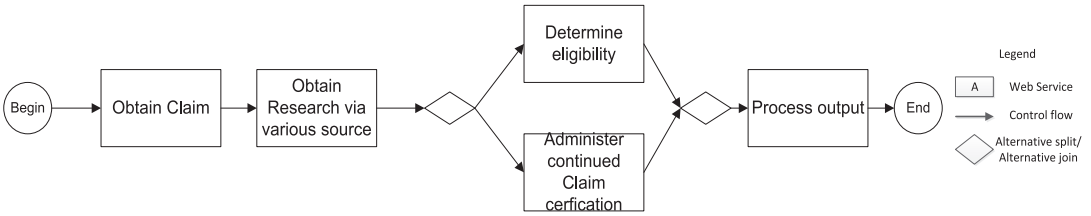
Application of the pattern: The pattern is used in process involving loan applications, new customer application, and originating or servicing mortgage loans.

(6) *Name*: Customer information management
Application domain: Insurance
Description: A pattern checks and completes customer applications.
The business process of the pattern:



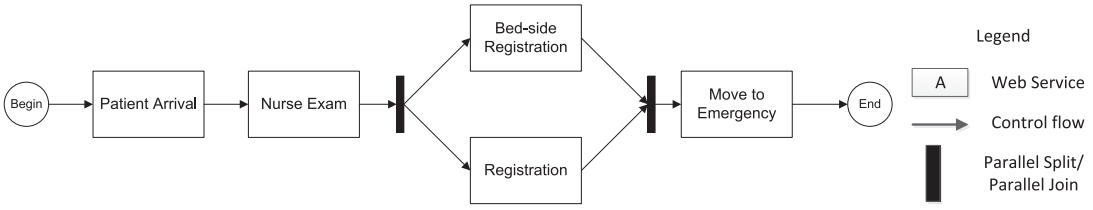
Application of pattern: The pattern is used in the process involving the validation and verification of customer application, claims submission, and claim processing.

(7) *Name*: Claim processing
Application domain: Government
Description: A pattern processes unemployment service claims.
The business process of the pattern:



Application of the pattern: The pattern is used in the process involving employer registration, employer contributions, and unemployment services administration.

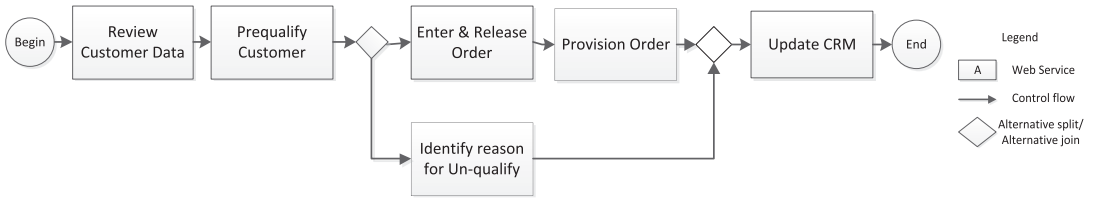
(8) *Name*: Emergency management
Application domain: Medical
Description: A pattern handles emergency services.
The business process of the pattern:



DOI: 10.1002/smr

Application of the pattern: The pattern is used in patient registration, triage, and evaluation processes. It includes subprocesses for blood pressure evaluation and blood tests.

(9) *Name*: Customer management
Application domain: Telecommunication
Description: A pattern deals with customer prequalification, credit check, and release of the order.
The business process of the pattern:



Application of the pattern: The pattern is used in digital subscriber line provisioning process that involves customer prequalification, credit check, release of the order, industry-generic credit review, and approval process.

ACKNOWLEDGMENT

We would like to thank Ms. Joanna Ng, Mr. Leho Nigul, and Ms. Janet Wong from IBM Canada for their helpful comments on this work.

REFERENCES

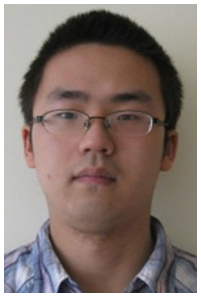
1. Lu J, Yu Y, Roy D, Saha D. Web service composition: a reality check. 8th International Conference on Web Information Systems Engineering, Nancy, France, December 3–7, 2007.
2. Andrews T, Curbera F, Dholakia H, Goland Y, Klein J, Leymann F, Liu K, Roller D, Simith D, Thatte S, Trickovic I, Weerawarana S. Business Process Execution Language for Web services. (Available from: http://www.ibm.com/developerworks/library/specification/ws-bpel/), last accessed on February 28, 2011.
3. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software*. Professional Computing Series. ISBN 0-201-63361-2.
4. Dijkman R, Dumas M, Garc´ıa-Ba˜nuelos L. Graph matching algorithms for business process model similarity search. In Proc. of BPM 2009, Ulm, Germany, Sept 2009.
5. Agrawal R, Gunopulos D, Leymann F. Mining process models from workflow logs. Sixth International Conference on Extending Database Technology, 1998; 469–483.
6. Cook JE, Wolf AL. Event-based detection of concurrency. Proceedings of the Sixth International Symposium on the Foundations of Software Engineering, 1998; 35–45.
7. Schimm G. Generic linear business process modeling. Proceedings of the ER 2000 Workshop on Conceptual Approaches for E-Business and The World Wide Web and Conceptual Modeling.
8. van der Aalst WM, van Dongen BF, Herbst J, Maruster L, Schimm G, Weijters AJ. Workflow mining: a survey of issues and approaches. *Data & Knowledge Engineering* 2003; **47**:237–267.
9. van der Aalst WMP, Weijters AJMM, Maruster L. *Workflow mining: which processes can be rediscovered?* BETA Working Paper Series, WP 74. Eindhoven University of Technology: Eindhoven, 2002.
10. Weijters AJMM, van der Aalst WMP. Process mining: discovering workflow models from event-based data. Proceedings of the 13th Belgium–Netherlands Conference on Artificial Intelligence, 2001; 283–290.
11. Weijters AJMM, van der Aalst WMP. Rediscovering workflow models from event-based data. Proceedings of the 11th Dutch–Belgian Conference on Machine Learning Benelearn 2001, 93–100.
12. Tang R, Zou Y. An approach for mining web service composition patterns from execution logs. 2010 IEEE Symposium on Web Systems Evolution (WSE), 2010.
13. Tang R, Zou Y. An approach for mining web service composition patterns from execution logs. The 8th International Conference on Web Services (ICWS 2010), Work in Progress Track, Miami, Florida, USA, July 5–10, 2010.
14. IBM WebSphere Process Server. (Available from: http://www-01.ibm.com/software/integration/wps/), last accessed on February 28, 2011.
15. Microsoft Azure Services. (Available from: http://www.microsoft.com/windowsazure), last accessed on February 28, 2011.
16. Google App Engine. (Available from: http://appengine.google.com), last accessed on February 28, 2011.
17. SourceLabs' Byron Sebastian joins Heroku as CEO. (Available from: http://venturebeat.com/2009/10/14/sourcelabs-byron-sebastian-joins-heroku-as-ceo/), last accessed on February 28, 2011.

DOI: 10.1002/smr

18. Agrawal R, Imielinski T, Swami AN. Mining association rules between sets of services in large databases. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., United States; 207–216.
19. Web Services Description Language (WSDL). (Available from: http://www.w3.org/TR/wsdl), accessed on February 19, 2011.
20. Porter MF. An algorithm for suffix stripping. *Program* 1980; **14**(3):130–137.
21. Miller GA. WordNet: a lexical database for English. *Communications of the ACM* **38**(11):39–41. DOI: 10.1145/219717.219748
22. advanced sample modeling projects for WebSphere business modeler. (Available from: http://www-01.ibm.com/software/integration/wbimodeler/advanced/library/samples612.html), last accessed on February 28, 2011.
23. SoapUI. (Available from: http://www.soapui.org/), last accessed on February 28, 2011.
24. Gschwind T, Koehler J, Wong J. Applying patterns during business process modeling. In Proceedings of the 6th International Conference on Business Process Management, Milan, Italy, 2008.
25. van der Aalst WMP, ter Hofstede A, Kiepuszewski B, Barros A. Workflow patterns. *Distributed and Parallel Databases* 2003; **14**(1): 5–51. DOI: 10.1023/A:1022883727209
26. Liang Q, Chung J, Miller S, Ouyang Y. Service pattern discovery of web service mining in web service registry-repository. IEEE International Conference on E-Business Engineering, 286–293.
27. Greco G, Guzzo A, Manco G, Pontieri L, Sacca' D. Discovering expressive process models by clustering log traces. *IEEE Transactions on Knowledge and Data Engineering* 2006; **18**(8). DOI:10.1109/TKDE.2006.123
28. Bose RPJC, van der Aalst WMP. Context aware trace clustering: towards improving process mining results. Symposium of Discrete Algorithm (SDM-SIAM), 2009; 401–412.
29. Di Francescomarino C, Marchetto A, Tonella P. Reverse engineering of business process exposed as Web applications. European conference on software maintenance and reengineering, 2009.
30. Kokash N. A comparison of web service interface similarity measures. *Frontiers in Artificial Intelligence and Applications* 2006; **142**:220-231.
31. Rahm E, Bernstein PA. A survey on approaches to automatic schema matching. *VLDB Journal* 2001; **10**(4). DOI: 10.1007/s007780100057
32. Dong X, Halevy A, Madhavan J, Nemes E, Zhang J. Similarity search for Web services. In Proceedings of the Thirtieth international Conference on Very Large Data Bases – Volume 30 Toronto, Canada, 2004.
33. Nayak R. Data mining in Web services discovery and monitoring. *International Journal of Web Services Research* 2008; **5**(1):63–81.
34. Liu W, Wong W. Web service clustering using text mining techniques. *International Journal of Agent-Oriented Software Engineering* 2009; **3**(1): 6–26. DOI: 10.1504/IJAOSE.2009.022944
35. Mahbub K, Spanoudakis G, Zisman A. A monitoring approach for runtime service discovery. *Automated Software Engineering* 2011; **18**(2):117–161.
36. Spanoudakis G, Zisman A. Discovering services during service-based system design using UML. *IEEE Transactions on Software Engineering* 2010; **36**(3):371–389.
37. Dijkman R, Dumas M, Garcia-Banuelos L, Kaarik R. Aligning business process models. In Proceedings of the 2009 IEEE International Enterprise Distributed Object Computing Conference (edoc 2009) (EDOC '09). IEEE Computer Society, Washington, DC, USA, 45–53. DOI: 10.1109/EDOC.2009.
38. Nejati S, Sabetzadeh M, Chechik M, Easterbrook S, Zave P. Matching and merging of statecharts specifications. In Proceedings of the 29th International Conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 54–64. DOI:10.1109/ICSE.2007.50
39. Minor M, Tartakovski A, Bergmann R. Representation and structure-based similarity assessment for agile workflows. In Proceedings of the 7th International Conference on Case-based Reasoning: Case-based Reasoning Research and Development (ICCBR '07).
40. Corrales JC, Grigori D, Bouzeghoub M. BPEL processes matchmaking for service discovery. OTM Conferences (1), 2006; 237–254.
41. Brockmans S, Ehrig M, Koschmider A, Oberweis A, Studer R. Semantic alignment of business processes. In Proc. ICEIS (3), 2006; 191-196.
42. Schäfera T, Knappa A, Merz S. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science* 2001; **55**(3):357–369.
43. Puhlmann F, Weske M. Using the pi-calculus for formalizing workflow patterns. In *Business Process Management*. Berlin, Heidelberg, 2005; 153–168.
44. Stefansen C. Expressing workflow patterns in CCS, 2005. (Available from: http://www.stefansen.dk/workflowpatterns.pdf).
45. Bultan T, Su J, Fu X. Analyzing conversations of Web services. *Internet Computing, IEEE* 2006; **10**(1):18–25. DOI: 10.1109/MIC.2006.1
46. Bultan T, Fu X, Su J. Analyzing conversations: realizability, synchronizability, and verification. Presented at Test and Analysis of Web Services, 2007; 57-85.
47. Di Lucca GA, Di Penta M, Fasolino AR, Tramontana P. *Supporting Web Application Evolution by Dynamic Analysis*. IWPSE: Washington, DC, US, 2005; 175–186.
48. Di Lucca GA, Fasolino AR, de Carlini U. *Recovering Use Case Models from Object-Oriented Code: A Thread-based Approach*. WCRE, 2000; 108–117.
49. Hamou-Lhadj A, Braun E, Amyot D, Lethbridge T. *Recovering Behavioral Design Models from Execution Traces*. CSMR, 2005; 112–121.

AUTHORS' BIOGRAPHIES

Bipin Upadhyaya is currently studying his PhD in the Department of Electronics and Computer Engineering at Queen's University, Canada. He has completed his MASci degree in Information Systems from Kookmin University Seoul, South Korea and his undergraduate from Tribhuvan University, Nepal. His research area includes service-oriented architecture, end-user service composition, and cloud-computing.



Ran Tang is a Software CAD Engineer at PMC (NASDAQ:PMCS). He received his BEng in Information Security Engineering from Sichuan University in 2008 and his MASci degree in Software Engineering from Queen's University in 2010. His research interests include Web service and service-oriented architecture.

Ying Zou received her BEng degree from Beijing Polytechnic University, her MEng degree from the Chinese Academy of Space Technology, and her PhD degree from the University of Waterloo in 2003. Currently, she is an associate professor in the Department of Electrical and Computer Engineering at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada Lab. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture.

DOI: 10.1002/smr