An Approach for Mining Web Service Composition Patterns from Execution Logs

Ran Tang and Ying Zou
Department of Electrical and Computer Engineering
Queen's University
Kingston, Canada
{ran.tang, ying.zou}@queensu.ca

***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 due to 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 log. A case study shows that our approach can effectively detect service composition patterns.**

Keywords-component; web service; composition; pattern mining; log

I. INTRODUCTION

A service-oriented application is composed of multiple web services to provide complex functionality that cannot be achieved by single web service. Service-Oriented Architecture (SOA) developers discover [22] desired web services and compose them into a service-oriented application using specification languages, such as Web Services Business Process Execution Language (WSBPEL) [1]. 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 web services and their control flows. The control flow defines the execution order in which the services should be invoked (e.g., sequential order, parallel order or alternative order). The web services contained in a pattern is frequently executed together by service-oriented applications in the defined execution order.

The service composition patterns reflect the best practices. They are well tested by large amount of adoptions. Therefore, an application potentially has better quality when it is composed by reusing such patterns. Service composition patterns can also help improve existing applications. By comparing the existing application against the patterns, the maintainer can identify the discrepancy and consider applying the patterns in the application. Web services used in patterns are more heavily used than other 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 services used in a pattern.

To facilitate the discovery and the use of service composition patterns, a great amount of research effort has been devoted in identifying service composition patterns. The research can be divided into top-down and bottom-up approaches. In the top-down approach [18], 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. In the bottom-up approaches [2][3][4][5][6][7][10], 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.

The remainder of this paper is organized as follows. Section II discusses the related work. Section III gives an overview of our approach. Section IV presents the approach of identifying a set of services commonly used together. Section V discusses the approach of identifying control flows among the services in the pattern. Section VI presents our case study. Finally, Section VII concludes the paper and explores the future work.

II. Related Work

In the business process management domain [17] [19], a composition pattern catalogs different forms of service interactions through control flows (e.g., sequential pattern, and loop pattern). Such pattern does not refer to a particular existing service. In our work, we identify service composition patterns with web services and the control flows among the services. Hence, the identified patterns are more concrete and ready to use for composing or improve applications. Dong et al. [12] propose an approach to find patterns between web services using interface matching. This approach identifies a pair of related web services by examining similarity of their outputs/inputs. The approach is limited to identify a pair of services with matched interfaces. However, the matched services may not have semantic relations. Our work identifies the services that are strongly associated in the service execution by analyzing historical execution logs.

Liang et al. [11] mine service association rules from service transactional data. Their approach aims to discover a binary association relation between two service sets. Agrawal et al. [2] propose a technique for mining workflows from execution logs. This approach builds a dependency graph using the order in which activities are recorded in the historical log. For example, the activity A depends on another activity B if A can only happen after B. However, their work did not further identify control flow structures based on the dependency graph. Cook and Wolf [4] develop a similar approach to recover business processes from event logs. Their 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. [5] propose an approach to recover workflows using Petri net theory. Similar to Agrawal [2], Aalst et al. use tasks in the workflow logs to construct Petri net to indicate the relationship among tasks. Aalst et al. can identify dependency relations, non-parallel relations and parallel relations to describe the execution order among tasks. In a heuristic approach [10], metrics are used to construct workflows using logs. Specifically, a dependency/frequency table is constructed based on 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. [24] mine multiple variant of process to represent different usage scenarios. Bose and van der Aalst [26] propose an approach to cluster instances to handle less structured process model. Di Francescomarino et al. [25] develop an approach to recover business process of web application 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: 1) 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 [4, 5]. 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 extends the aforementioned technique by using 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. 2) Different from the existing approaches which focus on recovering a single entire process, we mine the frequently executed patterns used in multiple service-oriented applications. Similar to the existing approaches [2, 5], we use the techniques to recover control flow of the pattern.

III. Overview of Our Approach

Fig. 1 gives an overview of our approach which is broken down into three major steps: (1) collect and pre-process execution logs; (2) identify a set of associated web services that are frequently executed together; and (3) recover the control flows among these services.

Collecting and pre-processing execution logs: The composed application is deployed and executed in a runtime execution environment, such as IBM WebSphere Process Server [16]. Traditionally, applications are individually deployed on proprietary infrastructure owned by the organization which 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 [13] and Google App Engine [15]. For instance, a platform named Heroku [14] hosts over 45,000 applications in 2007. Compared to 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. Each execution instance is uniquely identified with an identifier (i.e., id). 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 the 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 service invocation events. In particular, an ENTRY event is triggered when a service is invoked. An EXIT event occurs

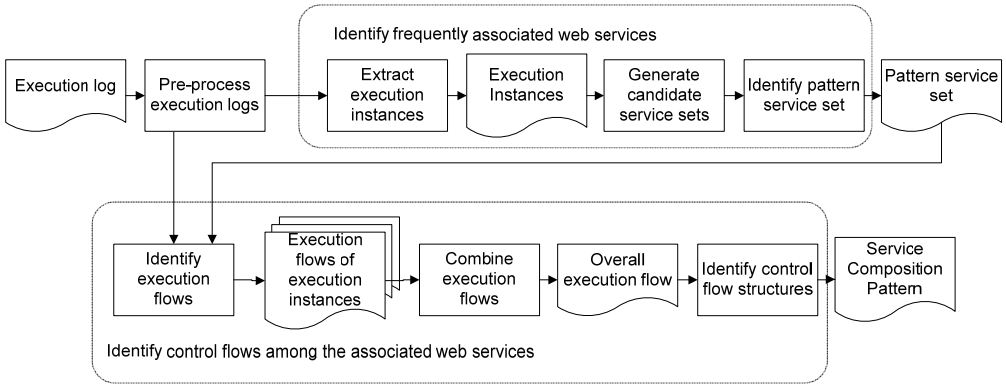


Figure 1 Overview of our approach

when a service completes the computation and returns results. Each event is recorded with the time of triggering, the name of the service which triggers the event, and the id of the execution instance and the underlying application.

Our work uses the logging facility built in the execution engine to record the particular types of events. However, it does not allow specifying the types of events to record. All the events are captured in the execution logs. We need to process the logs to extract the service invocation events.

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 the service set for a service composition pattern (i.e., pattern service set). The number of services in a service composition pattern is important. We propose that a service composition pattern contains at least four services. However, the required number of services may vary in different settings. A service composition pattern with very few services (e.g., two services) may provide incomplete functionality with minimal value for reuse. We generate candidate service sets and select the one with highest frequency as the intermediate pattern service set. From the intermediate pattern service set we construct the final pattern service set.

Recovering the control flow among the services identified as a pattern: The control flows among services would make the patterns more structured and ease the reuse of a 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, parallel structure) from the overall execution flows.

IV. IDENTIFICATION OF FREQUENTLY ASSOCIATED SERVICES

In this section, we discuss our approach to identify a set of frequently associated web services (i.e., a pattern service set). In our approach, we use Apriori [8] algorithm and develop a set of heuristic rules.

A. Extracting Execution Instances from Execution Logs

We analyze the execution logs to extract execution instances which 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. Each letter represents a web service. In a real application environment, different services may have same name. They can be differentiated using the URI of their service endpoint. It is also possible that a service is invoked more than once (e.g., in a loop in instances 7) in one execution instance. The multiple occurrences of a service should not affect the technique since we do not consider the number of occurrences or the order of services to determine the associated services.

TABLE I. EXAMPLE EXECUTION INSTANCES

Execution Instance ID	App ID	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, Y, Z, D}
6	3	{A, X, Y, Z, D}
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}

B. Generating Candidate Service Sets

We enumerate possible combinations of services with varying sizes to form candidate service sets. From the candidate service sets, we construct the pattern service set which contains the services in a service composition pattern. More specifically, we generate candidate service sets starting from size 1, and then increasing to size 2 and so forth. A web 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 the candidate service set. For example shown in Table I, the candidate service set, {D}, is invoked by 8 execution instances.

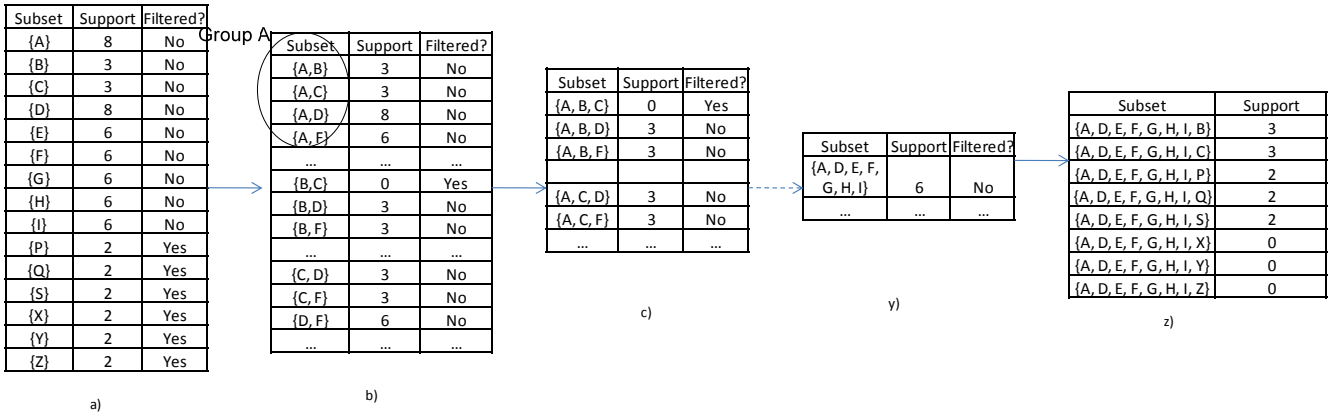


Figure 2 Example of generating candidate service sets

Therefore, the support of {D} is 8. The supports of all candidate service set of size 1 are shown in Fig. 2.a.

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

1) Collect candidate service sets that have the first n-1 identical services and only one different service to form one group. For example shown in Fig. 2.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 due to their identical first (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 the same as the initial sets of size n. The n-th and (n+1)-th services are pair-wise combinations of the n-th services of the initial set of size n. For example shown in Fig 2.b, the 2nd services in Group A are B, C, D and F. The pair-wise combinations of the 2nd services are BC, BD, BF, CD, CF, and DF. These combinations become the 2nd and the 3rd 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 the new candidate service sets of size 3 are listed in Fig. 2.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 is possible to 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 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 Fig. 2.

$$\text{threshold} = \frac{\#\,\text{execution instances}}{\#\,\text{service - oriented apps}} \qquad (1)$$

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 sets of size greater than 4 should have a support value no more than H. If no candidate service set with size m (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 based on 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 example shown in Fig. 2, 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.

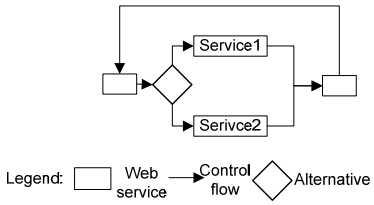


Figure 3 Alternative structure in service-oriented application

6) Identify a larger pattern service set by merging the candidate service sets that contains a common subset (i.e., the intermediate pattern service set S in step 5) and an additional different service. For example, 8 candidate service sets containing S are shown in Fig. 2.z and we

detect 2 sets from them to merge. Step 6 is used to handle alternative structures in the control flow of the service-oriented application as shown in Fig. 3. Suppose we have the candidate service sets S1 (i.e., S1= S ∪ {Service1}) and S2 (i.e., S2 = S ∪ {Service2}). We use S_{merge} (i.e., S_{merge} = S ∪ {Service1, Service2}) as the final pattern service set 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 $S_{mergeded}$) satisfy equation (2).

$$support(S) = support(S1) + support(S2) - support(S_{merged}) \quad (2)$$

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 double counting of such execution instances.

For example, {A, D, E, F, G, H, I} from Fig. 2.y is the intermediate pattern service set S in step 5). We further check its expanded candidate service set in Fig. 2.z. We find that the sum of 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.

Using the aforementioned steps, we identify a final pattern service set. We also identify the execution instances that contribute to the pattern. Specifically, 1) if no merging is archived 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. 2) 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. We use such execution instances to infer control flow of the pattern in next section.

V. Recovery of Control Flows of a Service Composition Pattern

In this section, we discuss our approach to recover control flows among services in a service composition pattern. To identify control flows, we analyze the execution instances that contribute to the pattern service set to identify their execution flow. An execution flow is a graph in which the nodes represent web services and the edges denote service ordering relations. Examples of execution flow for execution instances can be found in Table II. We combine 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.

The execution instances that contribute to the pattern identified in last section are shown in Fig. 4. Instance 5 and 6 do not contribute to the pattern and they are not included in the figure. 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.

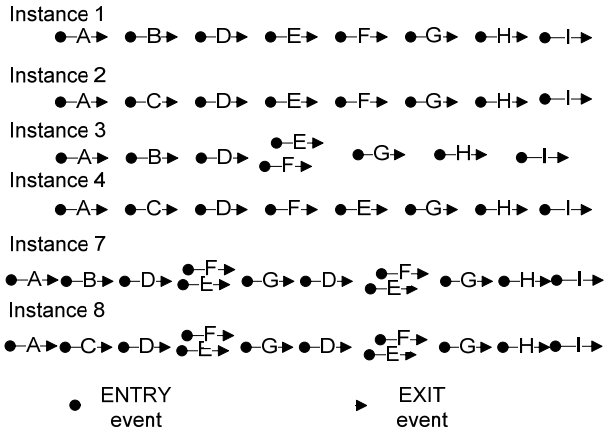


Figure 4 The execution instances with timeline of service invocation events

A. *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.

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 the service A to the successor service B (i.e., A→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: (1) the ENTRY B event occurs after the EXIT A event; (2) No web 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 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, Fig. 5 shows an event graph constructed using instance 7 from Fig. 4. 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 B represents that the event B immediately follows the 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 example shown in Fig. 4, the precedence relation between service A and service B (i.e., A→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.

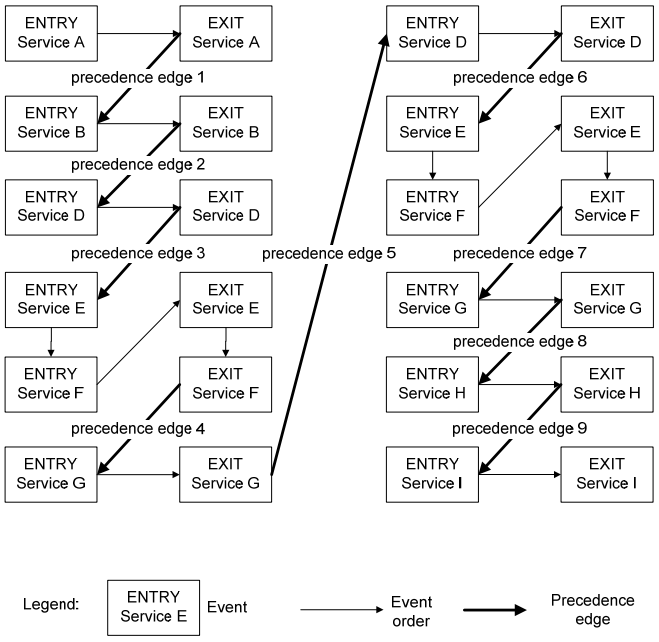


Figure 5 Identifying precedence and parallelism using event graph

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 example shown in Fig. 4, 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. In example shown in Fig. 4, precedence edge 4 has a destination service set containing service G.

In non-concurrent service execution, 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 execution, the source service set and destination service set may have more than one service. A precedence edge may indicate multiple precedence relations. In example shown in Fig. 4, from precedence edge 4, we can infer two precedence relations, namely, E→G and F→G.

Using the aforementioned approach, we construct an event graph for each execution instance shown in Fig. 4. The identified precedence relations for each instance are shown using arrows in Table II.

Iterative Relation: A loop exists in the execution flow graph when several services are repeated in the 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 event graph. For example in Fig. 5, the precedence edge 5 indicates a precedence relation G→D, which is actually an iterative relation. To distinguish an iterative relation from precedence relation, we use 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 current node, it indicates the entry service of a loop. We identify such an edge as an iterative relation. For 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→D).

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. Parallelism relation is also identified through 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. Similarity, parallelism relations can be identified in the destination service set. In example shown in Fig. 5, we infer a parallelism relation, namely E—F from the two services E and D, in the source service set of precedence edge 4.

Using this approach, we identify parallelism relations in each execution instance shown in Fig. 4. The results listed in Table II shows that parallelism is captured in the execution instances 3, 7, and 8 because the services are concurrently executed as shown in Fig. 4. This technique may not detect all parallelism because some parallel services can execute in sequence. We address this by merging execution flows.

B. 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 web services in the pattern. For example, by merging all execution flows listed in Table II, the overall execution flow shown in Fig. 6 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 bi-directional 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 having precedence relations because they are not concurrently executed. We replace such bidirectional precedence relations with a parallelism relation. In Table II, two precedence relations will connect service E and service F (i.e., E→F and F→E)

after merging instances 2 and 4. We replace them with a parallel relation between services E and F because E and F can be executed in 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 having a precedence relation when they are not concurrently executed in some execution instance. 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 execute before service F in this particular execution instance. The parallel relation prevails.

TABLE II. EXAMPLES OF 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

C. 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 graph 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.

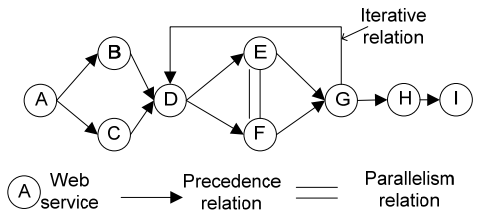


Figure 6 An example overall execution flow

Sequential structure: when a service emanates an edge denoting a precedence relation, this service is in sequential structure with its successor service. For example shown in Fig. 6, services H and I are in sequential structure.

Loop structure: Services in a loop structure can be repeated in an execution instance. We identify loop structure using iterative relation detected in previous subsection V.A. An edge representing iterative relation emanates from the exit service and leads to the entry service of a loop structure. For example shown in Fig. 6, the iterative relation between services G and D (i.e., G→D) indicates that the services along the path from D to G are in a loop structure.

Parallel structure: In 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 example shown in Fig. 6, the execution path along services D, E, G (i.e., D→E→G) and the execution path along services D, F, G (i.e., D→F→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.

Alternative structure: Multiple possible execution paths are executed under a certain condition, i.e., 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 paths are not identified as in a parallel structure (i.e., no parallel relations exist between different paths), we identify it as in an alternative structure. For example shown in Fig. 6, the path along services A, B, D (i.e., A→B→D) and the path along services A, C, D (i.e., A→C→D) are recognized as an alternative structure.

Using this approach, the control flow structures can be inferred from in the overall execution flow. To ease the reuse of the service composition patterns in the practices, we use BPEL to represent the identified service composition patterns along with the control flow structures. As an example, control flow structures are identified from the overall execution flow shown in Fig. 6 and the resulting service composition pattern is depicted in Fig. 7.

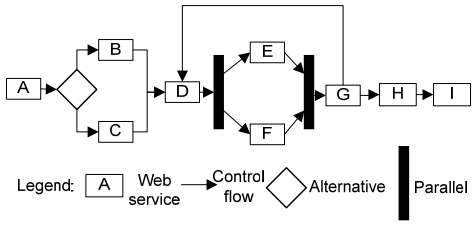


Figure 7 Identified service composition pattern with control flow structures

VI. CASE STUDY

To demonstrate the effectiveness of our approach, we conduct a case study to: 1) verify if our approach can accurately recover the control flow of a service-oriented

application from execution logs; and 2) validate if our approach can accurately identify service composition patterns.

A. Setup

In the case study, 74 service-oriented applications are studied. The applications are based on a set of reference business processes provided by a business process management architect from our industrial collaborator [9]. An overview of these service-oriented applications is listed in Table III. The applications specify activities and procedures used in various sectors. For example, the financial management applications manage financial policies, procedures, account and asset.

TABLE III. BUSINESS PROCESSES USED IN CASE STUDY

Industry	#	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.

The execution logs are collected from the testing environment when testing the 74 web services applications. The execution engine uses WebSphere Process Server installation. IBM WebSphere Process Server [16] is used to host the application.

In the process of testing, SoapUI [23], a standard SOAP-based invocation tool is used to automatically execute the service-oriented applications for a large number of times. The execution logs are recorded through the Common Event Infrastructure (CEI) built in the IBM WebSphere Process Server.

B. 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 verified by comparing with the original business processes. Specifically, the nodes and edges of the recovered control flow are manually compared with the original control flow. 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 73 out of 74 service-oriented applications listed in Table III. However, our approach fails to handle a special case in which an alternative structure has two branches with identical services in different order. Specifically, in the first branch, the service A executes before service B. In the other branch, the service B executes before service A. As a consequence, A and B appear in the execution log in random order. Our approach misidentifies A and B as in parallel structure since we use the random execution order as one criterion to identify parallel structure.

C. Results of Identifying Service Composition Pattern

To verify if our approach can correctly identify service composition patterns from the 74 service-oriented applications, we use a testing environment with varying numbers of execution instances for the applications and apply our approach. We measure correctness of the identified pattern service set and the correctness of the identified pattern control flow.

1) Correctness of the identified pattern service set. We use two metrics to measure the correctness of the identified pattern service set, i.e., precision and coverage.

Precision is defined in equation (3). It measures whether our approach can correctly identify the execution instances that contribute to a pattern service set. We manually verify if our approach correctly filter out the execution instances without patterns. Specifically, we manually verify whether each service-oriented application contributes to the pattern. Then we can determine whether the execution instances produced by such applications contribute to the pattern. The precision should be high if our approach is correct.

$$precision = \frac{\#\ correctly\ identified\ instances}{\#\ identified\ execution\ instances} \quad (3)$$

Coverage is defined in equation (4). 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 functionality of the applications which generate the log. If we know the execution logs are collected from a set of applications that fulfill similar functionality, we could expect that the coverage be high. In contrast, if the execution logs are collected from a set of applications with diverse functionality, we expect the coverage to be low.

$$\text{coverage} = \frac{\#\,\text{instances that contribute to the pattern}}{\#\,\text{total execution instances}} \quad (4)$$

2) Correctness of the identified pattern control flow. We use accuracy to measure the correctness of the control flow of an identified pattern. It is defined as equation (5). Among all the execution instances that contribute to the 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.

$$\text{accuracy} = \frac{\#\ \text{correct execution instances}}{\#\,\text{execution instances that contribute to the pattern}} \quad (5)$$

The results are shown in Table IV. The high precision shows that our approach can accurately identify pattern service set. The coverage of 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 pattern control flow is high but not 100% accurate due to 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 parallel structure in some applications. However, in other applications, A and B are arranged as sequence (i.e., A→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.

TABLE IV RESULTS OF IDENTIFIED SERVICE COMPOSITION PATTERNS

#	Domain of pattern	# Total execution instances	# Execution instances identified to be related to the pattern	Accuracy of pattern service set		Accuracy of control flow
				%Precision	%Coverage	
1	Financial Management	15K	10,285	100%	68.6%	91.2%
2	Human Capital Management	16K	11,110	100%	69.4%	87.7%
3	Supply Chain Management	15K	9,778	100%	65.2%	85.9%
4	Banking	16K	10,413	100%	65.0%	92.6%
5	Insurance	15K	9,559	100%	63.7%	88.9%
6	Government	17K	12,423	100%	73.0%	90.3%
7	Retail	15K	10,230	100%	68.2%	92.6%

Table V gives a summary of the service composition patterns identified by our approach. For each pattern, its domain of application, its diagram and a short description is provided.

VII. CONCLUSION AND FUTURE WORK

In this paper, we present an approach for identifying service composition patterns from execution logs. To identify a pattern, we find a set of web services frequently executed together along with their control flows. Our approach facilitates the documentation and reuse of service composition patterns that are proven to be successfully applied in the practices. Such patterns can 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. In the future, we plan to verify our approach using the execution logs generated from more applications.

ACKNOWLEDGMENT

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

REFERENCES

[1] F. Curbera, et al. “Business process execution language for web services”, http://www.ibm.com/developerworks/webservices/library/ws-bpel, last accessed on June 7, 2010

[2] R. Agrawal, D. Gunopulos, and F. Leymann, “Mining Process Models from Workflow Logs”, Sixth International Conference on Extending Database Technology, 1998, pp. 469–483.

[3] W. M. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. Weijters, “Workflow mining: a survey of issues and approaches”, Data Knowl. Eng. 47, 2 Nov. 2003, pp. 237-267

[4] J.E. Cook, and A.L. Wolf, “Event-based detection of concurrency”, Proceedings of the Sixth International Symposium on the Foundations of Software Engineering , 1998, pp. 35-45.

[5] W.M.P. van der Aalst, A.J.M.M. Weijters, L. Maruster, “Workflow Mining: Which Processes can be Rediscovered?” BETA Working Paper Series, WP 74, Eindhoven University of Technology, Eindhoven, 2002.

[6] A.J.M.M. Weijters, W.M.P. van der Aalst, “Process mining: discovering workflow models from event-based data”, Proceedings of the 13th Belgium–Netherlands Conference on Artificial Intelligence, 2001, pp. 283–290

[7] G. Schimm, “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] R. Agrawal, T. Imielinski, and A. N. Swami. "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 pp. 207-216

[9] Advanced Sample Modeling Projects for WebSphere Business Modeler, http://www-01.ibm.com/software/integration/wbimodeler/advanced/library/samples612.html, last accessed on June 7, 2010

[10] A.J.M.M. Weijters, W.M.P. van der Aalst, “Rediscovering workflow models from event-based data”, Proceedings of the 11th Dutch-Belgian Conference on Machine Learning Benelearn 2001, pp. 93–100

[11] Q. Liang, J. Chung, S. Miller, Y. Ouyang, "Service Pattern Discovery of Web Service Mining in Web Service Registry-Repository," E-Business Engineering, IEEE International Conference on, pp. 286-293

[12] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, “Similarity search for web services”, In Proceedings of the Thirtieth international Conference on Very Large Data Bases - Volume 30 Toronto, Canada, Aug 31 - Sept 03, 2004

[13] Microsoft Azure Services , http://www.microsoft.com/windowsazure, last accessed on June 7, 2010

[14] SourceLabs’ Byron Sebastian joins Heroku as CEO http://venturebeat.com/2009/10/14/sourcelabs-byron-sebastian-joins-heroku-as-ceo/, last accessed on June 7, 2010

[15] Google App Engine, http://appengine.google.com, last accessed on June 7, 2010

[16] IBM WebSphere Process Server, http://www-01.ibm.com/software/integration/wps/, last accessed on June 7, 2010

[17] W. van der Aalst, A. ter Hofstede., B. Kiepuszewski., A. Barros, “Workflow patterns”, Distributed and Parallel Databases 14(1), 2003 pp. 5–51

[18] R. Dijkman, M. Dumas, and L. Garc´ıa-Ba˜nuelos, “Graph matching algorithms for business process model similarity search”, In Proc. of BPM 2009, Ulm, Germany, Sept 2009

[19] T. Gschwind, J. Koehler, and J. Wong, “Applying Patterns during Business Process Modeling”, In Proceedings of the 6th international Conference on Business Process Management, Milan, Italy, Sept 02 - 04, 2008

[20] IBM WebSphere Integration Developer, http://www-01.ibm.com/software/integration/wid/, last accessed on June 7, 2010

[21] Eclipse IDE for Java EE Developers, http://www.eclipse.org/downloads/moreinfo/jee.php, last accessed on June 7, 2010

[22] J. Lu, Y. Yu, D. Roy, and D. Saha, "Web Service Composition: A Reality Check", 8th International Conference on Web Information Systems Engineering, Nancy, France, December 3-7, 2007

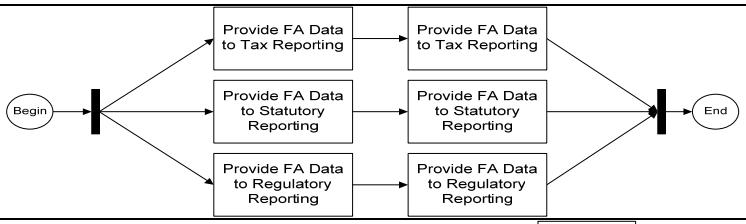
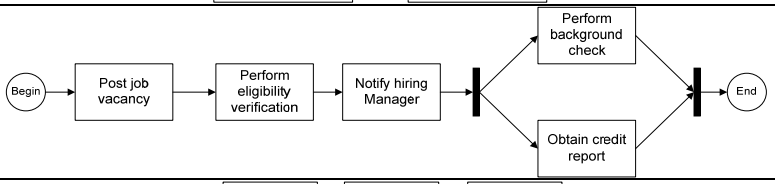
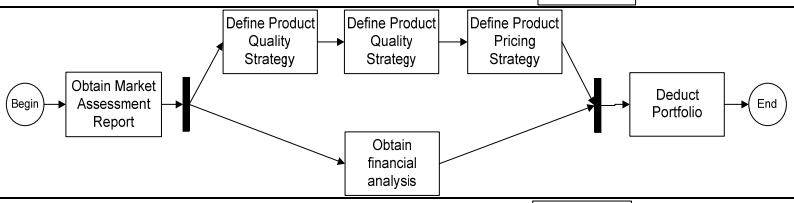
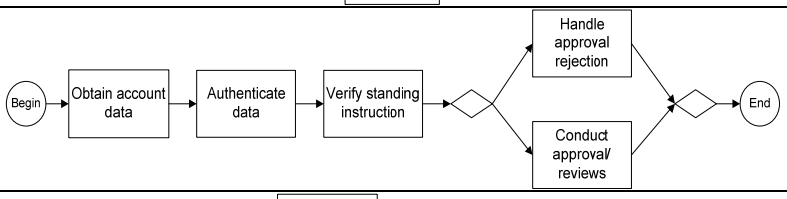
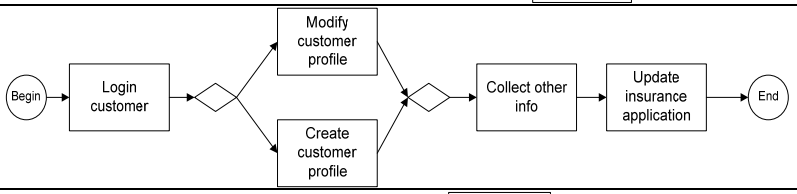
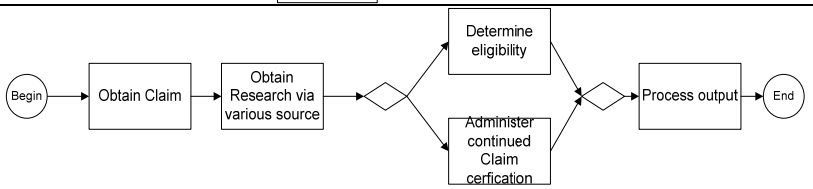
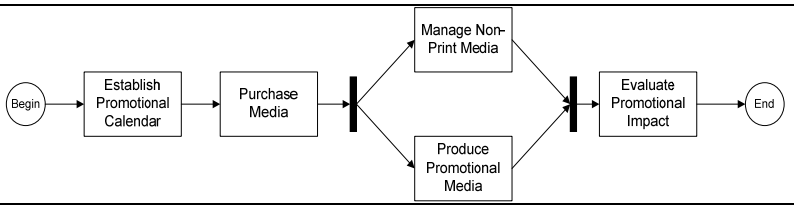
[23] SoapUI, http://www.soapui.org/, last accessed on June 7, 2010.

[24] G. Greco, A. Guzzo, G. Manco, L. Pontieri, D. Sacca'. Discovering Expressive Process Models by Clustering Log Traces. IEEE Transactions on Knowledge and Data Engineering, vol. 18 n. 8, 2006

[25] C. Di Francescomarino, A. Marchetto, and P. Tonella. Reverse engineering of business process exposed as web applications. European conference on software maintenance and reengineering, 2009

[26] R.P.J.C. Bose and W.M.P. van der Aalst. "Context Aware Trace Clustering: Towards Improving Process Mining Results" Symposium of Discrete Algorithm (SDM-SIAM) 2009

TABLE V SUMMARY OF THE COMPOSITION PATTERNS IDENTIFIED

#	Pattern domain	Pattern diagram	Pattern Description
1	Financial Management		A pattern shared by applications that provides the fixed asset data for tax, regulatory or statutory purposes.
2	Human Capital Management		A pattern shared by applications which handle staffing process.
3	Supply Chain Management		A pattern shared by applications that does the portfolio management of all the products and services offered by the organization to its customers.
4	Banking		A pattern shared by applications that process loan applications.
5	Insurance		A pattern shared by applications that check and complete customer applications.
6	Government		A pattern shared by application that process unemployement service claims.
7	Retail		A pattern shared by applicatoins that help retail store to promote products.