Resource Mining: applying process mining to resource-oriented systems. * **

Andrzej Stroiński, Dariusz Dwornikowski, and Jerzy Brzeziński

Institute of Computing Science, Poznań University of Technology
Piotrowo 2, 60-965 Poznań, Poland

{Andrzej.Stroinski, Dariusz.Dwornikowski,
Jerzy.Brzezinski}@cs.put.poznan.pl

Abstract Service Oriented Architecture is an increasingly popular approach to implement complex distributed systems. It makes it possible to implement complex functionality just by composing simple functionalities provided in form of services into so called business processes. Unfortunately, such composition of services may lead to some incorrect system behavior. In order dicover such depreciances and fix them, process mining methods may be used. Unfortunately, the current state of the art focuses only on SOAP-based Web Services leaving RESTful Web Service (resource-oriented) unsupported. In this article the relevance of adapting the Web Service Mining methods to new resource-oriented domain is introduced with initial work on process discovery in such systems.

Keywords: process mining, business process, logging, SOA, REST

1 Introduction

Currently, one of the most widely used approach to the implementation of distributed systems is Service Oriented Architecture (SOA). The approach aims to reduce costs of development and maintenance of information system. Additionally, the approach provides an easy integration among various heterogeneous information systems implemented accordingly to SOA. These benefits are made

** This work was supported by the Polish National Science Center under Grant No. DEC-2012/05/N/ST6/03051

possible by splitting simple system functionalities into independently developed applications called Web Services. Later on, composition of many Web Services into business processes is used to provide more complex functionality.

Nowadays, two different approaches to SOA are widely recognized [14]. The first one are SOAP-based Web Services, which are highly standardized, and use WSDL (Web Services Description Language) to describe their procedural interfaces and rely on SOAP (Simple Object Access Protocol) as their communication protocol. The second approach, introduced in [8] is REST (REpresentational State Transfer) and RESTful (resource-oriented) Web services, which take a declarative approach, are based on resources rather than functions, and use finite, known set of CRUD operations [15].

In both of the approaches however, the same problems with composition may yield incorrect system behavior, i.e. deadlock, livelock. In addition, number of composed and invoked services during system execution may be tremendous, making it hard to manage dependencies between them. In order to deal with this problem, a research of process mining (PM) may be used, i.e. process model discovery, verification and optimization [3]. Being a prominent and fast developing research area, PM has been also applied to SOA systems to discover process models from logs gathered from SOA services [7,6,4], improving Web Service behavior [1,9] and collecting logs [11]. As it can be seen process discovery, and generally PM, has been only applied to SOAP-based web services SOA systems. We believe that REST systems could also benefit from applying PM techniques. For that to be possible, one first needs to gather logs from a system, which are always the first step in every process mining method. There are papers that deal with gathering and collecting logs from SOA systems in order to apply PM techniques, or Web Service mining techniques. In [11] Authors tackle with the problem collecting event logs in order to extract process traces from application systems and integration portal log files. In [5] and [13] methods to deal with correlation of events with processes and processes instances are presented. The first article concerns formalization of interaction patterns in SOA to allow for assessment of existing systems in context of their ability to correlate event logs into processes and conversations, whereas the second one discusses different approaches to event correlation between interacting services and methods of correlation discovery. Unfortunatelly, authors are considering only the interactions between services without taking into account local events. Unfortunately, all the articles focus on SOAP-based systems, so the methods and techniques they present cannot be directly applied to resource-orinted systems (consisting of RESTful Web Services), due to different nature of SOAP-WS and REST. In this article we tackle the problem of adapting the Web Service Mining methods to RESTFul Web services domain. In addition, we introduce context logging, a technique of log enrichment in order to make possible to infer process related data in resource-orinted systems (Section 2). Furthermore, process and process instance reconstruction algorithm for resource-oriented systems based on approaches for SOAP-based services is presented (Section 2.1). We also propose and discuss a prototype framework implementation (Section 3). Finally, utiliza-

tion of proposed methods together with classic PM methods in order to achieve Resource Mining (RESTful Web Service Mining) is shown (Section 4).

2 Resource Mining: the resource-oriented approach to Web Service Mining

In order to discover process models in resource oriented distributed systems there is a need to adopt already existing methods of Web Service Mining and/or develop new ones in respect to differences between resource-oriented systems and SOAP-oriented. Unfortunately, because of lack standards like BPEL, WS-Addressing or WS-Coversation, the correlation patterns formalized in [5] are hard to fulfill. In addition also solutions presented in [13], are not sufficient for resource-orineted sytems. Main differences between both approaches are as follows: (1) In resource-oriented approach the concept of service is not the central one. It only serves as the application component which is composed of a callable set of resources. Not services but individual resources are important from a client's perspective, therefore individual resources need only to be considered. Next (2), in contrary to SOAP-based services execution state is stored on the client's side (active), not on the server side (resources). Thus, underlying process logic is executed in the client by operating on resources by invoking them using finite set of predefined CRUD operations. (3) Resources are passive, i.e. they only provide data representation and implementation of available operations. Executing them is only possible on client's demand. Such an approach introduces unique and often desirable properties: stateless communication and unified interface [8]. (4) Business process is a resource. In order to achieve complex functionality in a resource-oriented system, client must compose system resources invocations into so called workflow or business processes. Upon client's action, a passive resource may on behalf of that client act as client for other resources, we call it a *process resource*. (5) Resources are hierarchically dependent on each other, some of resource representations may be included in other resource representations. Correctly modeled and implemented resource-oriented system will use `URI`s to pinpoint such inclusion [15]. (6) HTTP protocol is used as communication layer, so the semantic of HTTP messages is used in order to determine request handling. In the case of SOAP-based services, HTTP protocol only serves as a simple transport layer to ensure delivery of SOAP messages. (7) Additional consequence of using HTTP protocol is a synchronous communication model, which guarantees receiving response for every request. In this article we assume only synchronous communication as a basis for further discussion about more complex communication patterns (sequence of synchronous interactions modeling asynchronous communication). (8) In contrary to SOAP, HTTP lacks one-way communication so there is no such type of communication in resource-oriented systems (standard request-response model). Next (9), there are no standards like WS-Addressing or WS-Conversation, so there is no support for using and logging process related informations like process IDs and process instance IDs. Finally, (10) process resource may be nested, what mean that each process resource may be further

orchestarte into more complex process resource. Because of that, process logic also may be nested in internal events at such resources. In consequence there is need to dicovering not only traces of process of communication events and correlete them into process instances, but also internal resource events that are available in the log.

The crucial problem in gathering logs in resource-oriented systems is the lack of appropriate logging level available in the current SOA implementations [6]. This problem occurs due to usage of application servers designed and developed exclusively for request-response model of interaction. In this model, server passively awaits for client requests. Upon receiving request, the server processes it and sends a response back to the client. As a result, only information about received message and returned response is stored in an event log. What is lacking is process related data, such as process instance id and process id. In addition, resources may act as clients and such events are usually not stored in log (difference (4)).

Next, there is a need to group events concerning each of the resources belonging to the particular RESTful service (difference (1)). Usually, services store invoked URI address in log so this information may be used, or if application server does not support such feature, a solution is presented in Section 3. Next, there is also a problem of handling resources by parallel instances. In the current application servers like Apache Tomcat, new instance of resource is created for each incoming request to its URI. In consequence, each of instances log information concurrently into a log file, so event log interleaving problem occurs (Figure 1).

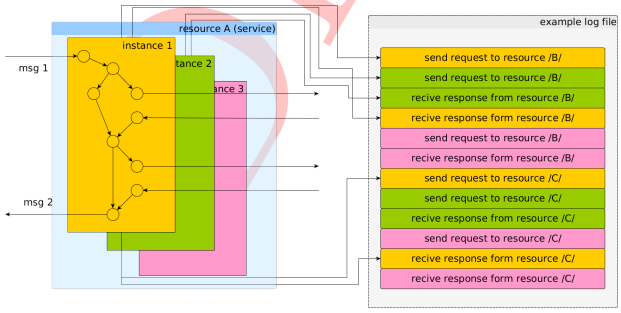


Figure 1: Interleaving of events in process log by multiple instances of resource

`Instance 1` of `Resource A` is invoked and sends a request to `resource B`, an appropriate log entry in log is stored. Next, `instance 2` is created upon second request to `resource A`. The `instance 2` request to `resource B` is sent, and response for that request is received (line 2 and 3 at Figure 1). Next, `instance 1` receives response and logs this information (line 4). As the example shows, if there is no information about instances in the log, that create log entry, there is no possibility to tie the receiving of message to sending it. In addition, log also

includes information about incoming messages like `msg 1` and returned messages `msg 2`. There is a need to correlate thes messages with each other and with outcoming messages, in order to associate them with proper service instances, as well as to keep log ordering relation [2]. This allows to discover local process of each resource (Figure 2b):

$$a \succ_L b \text{ if and only if there is trace where event } b \text{ immediately precedes event } a \quad (1)$$

This relation orders all local events of some resource (local process at Figure 2b). In order to deal with above problems we introduce context logging. The main concept is to add a unique ID (Context ID) of the resource instance to each logged event. As a result each service instance will add additional field to event log during logging called `context`. This context simply correlates incoming messages, with outgoing messages, and some local events. Such a context log allows to specify events that take place within different instances of resources allowing to generate an independent event log files for each resource in the service, and each instance of that resource (local log at Figure 2b). In addition, if we enforce adding local context as a additional HTTP header (it is possible because of difference (6)) it is also possible to correctly preserve ordering (correlation) relation introduced in [13] (*atomic correlation condition*) or in [5] (*reference-based correlation*) between interacting resources based on context information in HTTP header (Equation 2).

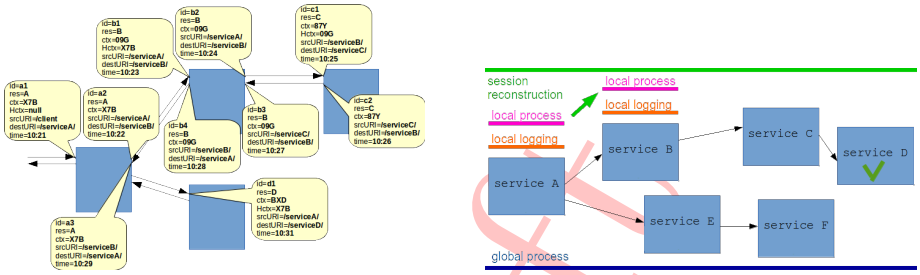
$$\begin{aligned} a \succ_{ctx} b \text{ if and only if there is trace in event log where } \#_{ctx}(a) = \#_{hctx}(b) \wedge \\ \wedge\ \#_{res}(a) = resourceA \wedge \#_{res}(b) = resourceB, \text{ where } resourceA, resourceB \in Res \wedge \\ \wedge\ resourceA \neq resourceB, \text{ where } Res \text{ is a set of all resources in the system, and} \\ \#_{ctx}(e) = A \text{ means value of field } ctx \text{ of event } e \text{ is A} \end{aligned} \quad (2)$$

These relations describe a situation where `resourceA` invokes `resourceB` and logs this information with $\#_{ctx}(a)$ label as event a and `resourceB` receives this message and logs this event with context label passed by `resourceA` ($\#_{hctx}(b) = \#_{ctx}(a)$) and with local context label $\#_{ctx}(b)$. Next step is to reconstruct session and a global process and generate appropriate `processID` and `instanceID`, basing on context information (session reconstruction and global process in Figure 2b). In order to reconstruct session one needs to apply information about which resource instance invokes other resource instance. Such information allows to retrieve the whole workflow information of interacting resources during business process execution. In order to achieve that, we ask each resource to send its local context in HTTP header to its callees. Then each callee needs to log this header as a receiving event log entry with its own local context. As a result, each of invoked service has information about local context within it was called. Based on the context ordering relation and partial context information, the algorithm for session reconstruction can be applied.

2.1 Simple process and instance reconstruction algorithm for resource-oriented systems

The main idea behind session reconstruction is to add appropriate `processID` and `instanceID` to events in the log. The main problem is to tie events with a process instance, i.e. process run. In our approach we are using the idea of

context logging from Section 2. We are assuming that each resource enriches its log entries with context field generated by each of its instances. This allows us to distinguish event log entries created by different resource instances (even if they are in the same log file). Next, if resource plays client role during process execution, it must include context information into its all outgoin messages. This ensure that context information is transfered to nearest neighbors, and allows to tie interacting resource instances with each other. We use HTTP header (`hctx`) to transfer context. The example of log entries generated by resources during interaction is presented in The Figure 2a.



(a) Log entries example (b) Idea and limitation

Figure 2: Context logging

Therefore, based on the relation in equation 2, and the obtained context log, we are construct a chain of connected resource instances. First, we generate a set `CTX` that contains information about the dependence among resource instances occurring in the log:

$$CTX = \{(e_1, e_2) \mid \exists_{e_1,e_2 \in L} \ \#_{hctx}(e_2) = \#_{ctx}(e_1)\} \tag{3}$$

We are looking a pair of events in the event log that represent communication between two resource. Such events in resource-oriented (RESTful) log are easy to find because they include additional data related to HTTP protocol (header, method etc.). The sender of message will include its local context into `hctx` header of message, so as a result the receiver will log in event e_2 the sender local context (it is included in HTTP header) next to its own local context. Into set CTX we put tuples of events between communication resources, where local context of first event (e_1) is equals to received form communication invoker context in event e_2. Next, we search the log for global process starting events (sent by process principal), according to:

$$F = \{f \mid \forall_{f \in L} \ \#_{hctx}(f) = null\} \tag{4}$$

Global process starting event is recognized be empty `hctx` HTTP header value ($\#_{hctx}(f) = null$). This occurs when $\#_{res}(f)$ resource is invoked by a process principal, because process principal is not a part of process so it does not include context information in invoke messages. Each event in F represents the initial event in global process execution so the size of set $|F|$ represents a number of process instances occurring in event log. Next, all so called context chains of

events are calculated (based at correlation condition in [13]). The context chain is an ordered set of events that represents context flow during process execution in one global process instance (one chain represents one global process instance).

$$\textbf{foreach } f_j \in F \textbf{ do } CHAIN_j = \{f_j, E_j, CTX_j\}$$
$$\text{, where } f_j \text{ is a starting event in this chain , and } E_j \text{ is a set of events in this j-th chain and}$$
$$CTX_j \text{is a set of context dependency between events in } E_j \cup \{f\}$$
$$\text{, where } j = 1...|F|(|F| \text{ is a number of process instances occurring in log}). \quad (5)$$

In order to do that, we need to find all events sets of context dependent events in each of the context chains (one context chain for each starting event):

$$E_j = \{e \mid (\forall_{e \in L} \exists_{e' \in L \wedge e' \in E_j} (e \succ_{ctx} e' \vee e' \succ_{ctx} e)) \wedge (\exists_{e \in L} f_j \succ_{ctx} e)\} \quad (6)$$

Set E contains events e, such that all events in this set are context dependent on at least one other event in this set, additionally at least one event from this set is context dependent on starting event f_j. Next, the set of context dependencies between events of set $E_j \cup \{f_j\}$ CTX_j is calculated as follows:

$$CTX_j = \{(e_1, e_2) \mid \exists_{e_1, e_2 \in E_j} \#_{hctx}(e_2) = \#_{ctx}(e_1)\} \quad (7)$$

Set CTX consists of tuples (e_1, e_2) where event e_2 is context dependent on event e_1. In consequence, each context chain shows mutually interacting process instancess in some (still unknown) global process. As a result, the process `instanceID` may be generated and added to each of context chains. Further, each event can obtain `instanceID` from context chain it belongs to. Unlike most of approaches, other events (not only communication events) must be added in order to take local processing of resources under consideration (differences (10)). This results in more accurate process models because sending messages may be dependent on some local resource event. This results in the $Instance$ set:

$$Instance_j = \{(f_j, E_j \cup \{e \mid \forall_{e \in L} \exists_{e' \in E_j} \#_{ctx}(e) = \#_{ctx}(e')\}, CTX_j, SUCCj)\} \text{ , where}$$
$$f_j \text{ is a starting event in chain } CHAIN_j \text{ and } E_j \text{ is a set of events in chain } CHAIN_j, \text{ and}$$
$$CTX_j \text{ is a set of context dependency occurring in this process instance,}$$
$$\text{and } SUCC_j \text{ is a local resource events ordering set} \quad (8)$$

$SUCC_j$ is a set of tuples showing local order relation among events of the same resource instance. It contain all the events belonging to the resources involved in $j-th$ context chain.

$$SUCC_j = \{(e_1, e_2) | \forall_{e_1, e_2 \in L} \exists_{e' \in CHAIN_j}$$
$$(\#_{ctx}(e1) = \#_{ctx}(e') \vee \#_{ctx}(e2) = \#_{ctx}(e')) \wedge (\#_{ctx}(e_1) = \#_{ctx}(e_2) \wedge$$
$$(\#_{res}(e_1) = \#_{res}(e_2) = r \wedge e_1 \succ e_2)\} \quad (9)$$

The final step is to determine which of the found instances belong to which process. The idea to discover processes, and correlate instances with them is based on differences (1) and (4) that everything is represented in form of resource. Even business process must be provided in form of resource callable by HTTP protocol operations. In resource-oriented systems, business processes are executed by invoking resources call other resources on behalf of the *process principal*. Therefore, the final step is to analyze resource property of each first log

entry ($\#_{res}(f_j)$) of each of $Insance_j$ in order to find such process resources. We are analyzing only the first event in each instance, as they are invoked by process principals ($\#_{hctx}(f) = null$), so they are the starting point of process execution. Next, for each unique resource (called *process resources*) we are generate `processID`, because each instance starting from the same resource is an instance of the same process resource. Therefore this allows us to correlate instances with process. Next, we are calculate sets of processes instances for each of process resources occurring in the log:

$$Process_n = \{i \mid \forall_{i_1,i_2 \in AllInstancesInLog}\ (\#_{res} first(i_1) = \#_{res} first(i_2) \wedge i_1 \neq i2) \oplus i_1 = i2)\}$$
$$\text{, where function } first() \text{ returns } f_j \text{ for } CHAIN_j \quad (10)$$

As a result the algorithm returns a set of $Process_n$ sets that include several $Instance_j$. Based on this, there is a need to review all events in event logs of all resources in the system, and add to them instanceID accordingly to ID of $Instance_j$ that this event belongs to. Then add `processID` accordingly to ID of $Process_n$ to which that event $Instance_j$ belongs to. As presented in Figure 2a there is only one global process ($Process_0 = \{Instance_0\}$) and one instance ($Instance_0 = \{\{a1\}, E_0, CTX_0, SUCC_0\}$, where $E_0 = \{a1, a2, a3, a4, b1, b2, b3, b4, c1, c2, d1, d2\}$, and $CTX_0 = \{(a2, b1), (b2, c1), (a3, d1)\}$, and $SUCC_0 = \{(a1, a2), (a2, a3) \ldots (d1, d2)\}$)

3 Non-invasive context logging for JSR-311 with AspectJ: a case study

As it has been shown, context logging can be used to differentiate between seperate resource instances in seperate process instances of multiple processes. The idea behind context logging is to inject HTTP headers into messages that pass through the system. This simple technique can be implemented in three ways: service instrumentation, proxy servers introduction, and semi non-invasive way.

We show that non-invasive logging is possible for a wide range of enterprise systems, i.e. Java based RESTFul Web services, implemented according to the JSR-311 standard [10]. We use AspectJ [12], a reference implementation of aspect oriented programming (AOP) language for Java virtual machine, and Apache Tomcat application server.

Jersey comprises to JAX-RS, a JSR-311 stanardized API of implementing RESTful Web services in Java. Both, the standard, and Jersey are widely used in a number of enterprise application servers and frameworks. We present a proof of concept implementation of our AspectJ context logger for RESTful systems implememented according to JSR-311, and in fact, this is our only technical requirement. Our approach will work for any java application server and implementation of JAX-RS. On the other hand, we believe that the same approach can be used for any other technology that offers support for AOP, such as .NET, Python or Ruby.

We take advantage of the fact that in JAX-RS (Jersey), every RESTful Web service needs to be defined in a class, annotated with certain decorators, e.g. `@Path`. The listing below shows an exemplary Web service implemented in Jersey.

`@Path("/hello")` annotation says that the Web service will be accessible under `"/hello"` URI resource. The method annotated with `@GET` and `@Path` handles every GET operation issued on `"/hello/world"` resource, @GET can be substituted with any other HTTP operation. `@Produces` or `@Consumes` in thecase of POST,PUT defines what content type the resource returns or accepts.

```
@Path("/hello") // every class has @Path
public class Hello {
    // every method has @OP annotation
    @Produces("text/html") @GET @Path("/world")
    public Resource handler(@Context HttpHeader headers, @Context HttpServletRequest request) {}
```

Thanks to the standardized API, a universal AspectJ context logger for incoming messages can be implemented in a quite simple way. One needs only to define pointcut, which catches every execution of any method placed in any class annotated with `@Path`, `@Produces` and `@GET`. In our implementation an advice is called when the pointcut is reached. First local context is generated, which in our case is the hash-code of a current object instance. Next, if the `HCTX` header is present in the request, it is stored and logged alongside with the local context, remote caller IP, HCTX value, and request URI taken from the `@Path`. Local context is then appended to every outgoing request from the current service instance in a `HCTX` header.

```
execution(@javax.ws.rs.GET @javax.ws.rs.Produces public * *(..)) &&
          within(@javax.ws.rs.Path *)
```

The situation gets more complicated when we want to catch and log messages sent from a service. In that case, not only we have to alter the outgoing message with context logging HTTP header `HCTX` but also the HTTP client call can be done in some arbitrary way. In our case study, we assume that these external calls are done from the same thread that handles the incoming request, i.e. synchronously. We also assume that JSR-311 client API is be used. Therefore, a pointcut can be defined to catch all calls to methods named `request` within classes annotated with `@Path`. Such an approach allows us to alter the request headers originating from the Web service, and thus pass the context to external Web services, according to context logging approach. Assuming that external Web services are also equipped with our aspect context logger, logs of all messages received and sent in the whole system can be created.

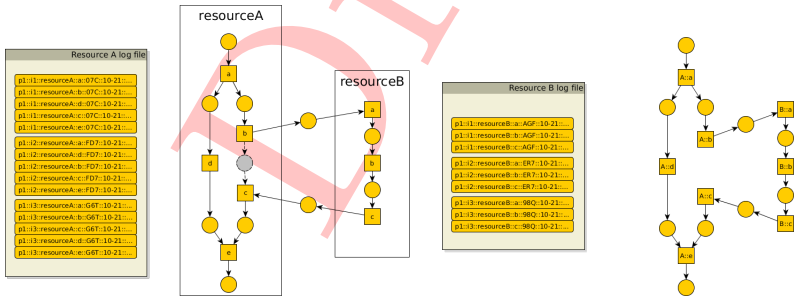
```
call(public * *.request(..)) && within(@javax.ws.rs.Path *)
```

There are some requirements we need to impose on how services are implemented with Jersey. We require that every method handling requests needs to return `Resource` object, and take the following arguments: `@Context HttpHeader` and `@Context HttpServletRequest`. This is needed to extract information, such as remote caller IP, request headers, and to inject our own header. Another difficulty we came across is the way the situation when service we equip with aspects calls some external service. In our approach we assumed that the call is done in the same thread as incoming request handling but this does not need to be the case. If it's not the case, it is still possible to implement a logger, by examining the call stack to determine how the current thread was called. By

comparing this with the call list kept in aspects, it is possible to determine which service instance was the original caller.

4 Applying alpha algorithm for resource-oriented context preprocessed log

After preprocessing the context log by the process and instance reconstruction algorithm, it is now possible to apply classic process mining algorithms. In our initial work on service mining in resource-oriented systems, we have used basic alpha algorithm [2]. It is a simple algorithm that allows to discover process model in form of a petri net. The idea behind this approach is to use simple ordering relation (Equation 1) that occurs in the event log file. Unfortunately, in the real case scenario in resource-oriented systems, it is very unlikely that each resource in the system will log into the same log file with respect to some global clock and with respect to some global ordering relation. Therefore the problem of gathering logs from distributed resources with respect to global ordering arises. In addition, the basic version of alpha algorithm does not take the resource perspective into account. So the first step is to make logs unique globally (usually events IDs are only unique locally at resource). Without distinction of resources, two events occurring at different resources may leave identical log entries, so as a global identifier is the concatenation of resource URI (it must be unique by the definition) and its local identifier (unique at the resource). Lets consider example shown in Figure 3.



(a) original process

(b) discovered process

Figure 3: Process model

The `resource A` is a process resource and invokes `resource B` during its execution. Next, two events with IDs `a` occurr in two different resources `resource A` and `resource B`. If we omit resource information they are indistinguishable form each other. In order to use basic alpha algorithm we need to flatten the log, to make sure ID of each event in the system is unique, we are concatenate resource URI and local ID — `A::a` and `B::a`. Alpha algorithm takes one log file

with multiple traces (instances) of exactly one process as its input. In order to use this method, first we need to gather distributed resource log files and concatenate them into one file for each process found by algorithm from Section 2.1. The first problem with concatenation, of independently generated log files, is the order in which concatenation is performed. Different approaches to deal with this problem have different impact on the results. It is because the alpha algorithm only uses flat order of events in a log file to determine if two events are executed in parallel, in some order, or are independent on each other. In consequence, simple concatenation of resource log files will result in violation of causal dependency of events. In the considered example (Figure 3a) adding `resourse B` log file at the and of `resource A` file will result in incorrect dependency relation between event `A::e` and `B::a`. This will lead to incorrect conclusion that these events are not independent. In order to deal with this problem there we need to perform another preprocessing phase of the event log in order to identify the communicing resources and appropriately concatenate event logs of subinvoked resources. We consider only synchronous communication so if a resource does not execute multiple parallel threads, all invocation events must be followed by corresponding response events (`A::b` and `A::c`). If there are two parallel threads, then all events in the second thread must be parallel to both the invocation and the response handling event (`A::d`). In order to concatenate log files and respect ordering relation among events in both resources (context dependency between two events in different resource `A::b` and `B::a`), and in addition to respect local ordering of events, we use previously calculated sets of context chains in Equation 5. For each chain, and for each resource instance occurring in context chain, we look for communication events invoking and handling response (*communication pair* $CP = (start, end, ctx_1, ctx_2)$). Each of such pairs consists of: *start* - starting event (invoking event), *end* - ending event (response handling event), ctx_A - context of invoking resource and ctx_B - context of invoked resource. Thanks to that, during pre-processing phase we put all events in the invoked resource event log file between the starting event and the ending event. Additionaly, not to disturb parallel relation of concurrent events there is a need to generate additional traces, not originally included in log file. Lets consider example in Figure 3. We search for all *communication pairs* in the log. The only found *communication pair* is: `A::b`, `A::c`, `A::07C` and `B::AGF`. Because we are dealing with synchronous communication, we add all events of resource instance `B::AGF` between the events `A::b` and `A::c` of resource instance `A::07C`. The problem occurs with event `A::d`, which is parallel to communication events in `resourceA`. In order to respect the parallel relation, we need to generate new traces (the minimal set of them) that will render all events in log file of `resourceB` to be also parallel to event `A::d`. To do that, we need to generate new process traces (instances) with respect to the following condition:

$$\forall_{CP \in Log_A}((e \mid\mid start \wedge e \mid\mid end) \implies (\forall_{f \in Log_B}(f \mid\mid e))$$
$$\text{, where } Log_A \text{ is invoking resource log and } Log_B \text{ is invoked resource log}$$
$$\text{, and } a \mid\mid b \Leftrightarrow a \succ_L b \wedge b \succ_L a \text{ , where } L \text{ is some log} \quad (11)$$

As a result, new traces are generated and we can execute the alpha algorithm for each process occurring in the log. The discovered petri net is shown at Figure 3b. In comparison to the original model in Figure 3a, the dotted places and arcs are not present. It is because, there is no longer relation between events `b` and `c` at `resource A` afer log preprocessing. This is a side effect of adding `resource B` log. In conclusion, presented example shows that alpha algorithm is able to mine processes based on a preprocessed resource-oriented log. Some drawback

of this approach is that during mining interaction between resources, some local dependencies are lost. In context of global process mining this is not an issue, because from a global point of view the workflow is in fact transfered to invoked resource.

5 Conclusions and future work

We have shown our initial work on applying Web Service Mining into a domain of resource-oriented systems. The approach shows how current methods must be adjusted in order to discover process models based on event log obtain from a RESTful system. Presented considerations leads to several conclusions and feature challenges. First, practical framework to obtain context enriched log concerns only the case where all resources are implemented in JAVA accordingly to JSR-311 standard. Unfortunately, in the case of resource implemented in different technology more work may be required. In future we would like to address this problem. Another direction of research is to develop algorithms dedicated resource-oriented systems that do not need to preprocess event log. This may lead to more accurate process models by using all information available in the log, like hierarchy relation along resources and/or message semantics. Finally, current process mining methods work only with global process. In our approach to reconstruct process related information we discover multiple process resources but later we execute process discovery algorithm for each of them separately. Our current work concerns developing methods for discovering processes models based on multiple process logs.

References

1. van der Aalst, W.: Service mining: Using process mining to discover, check, and improve service behavior. IEEE Transactions on Services Computing 99(PrePrints), 1 (2012)
2. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
3. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer Publishing Company, Incorporated, 1st edn. (2011)
4. van der Aalst, W., Verbeek, H.: Process Mining in Web Services: The WebSphere Case. IEEE Bulletin of the Technical Committee on Data Engineering 31(3), 45–48 (2008)
5. Barros, A.P., Decker, G., Dumas, M., Weber, F.: Correlation patterns in service-oriented architectures. Proceedings of the 10th international conference on Fundamental approaches to software engineering (2006)
6. Dustdar, S., Gombotz, R.: Discovering web service workflows using web services interaction mining. International Journal of Business Process Integration and Management (2007)
7. Dustdar, S., Gombotz, R., Baina, K.: Web services interaction mining (2004)
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)

9. Gaaloul, W., Bhiri, S., Godart, C.: Research challenges and opportunities in web services mining (2006)
10. Hadley, M., Sandoz, P.: Jax-rs: Java api for restful web services (2008)
11. Khan, A., Lodhi, A., Köppen, V., Kassem, G., Saake, G.: Applying process mining in soa environments. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave Workshops. Lecture Notes in Computer Science, vol. 6275, pp. 293–302 (2009)
12. Kiczales, G., Hilsdale, E., Hugunin, J., at. al: An overview of aspectj. In: Proceedings of the 15th European Conference on Object-Oriented Programming. pp. 327–353. ECOOP '01, Springer-Verlag, London, UK, UK (2001)
13. Motahari-Nezhad, H.R., Saint-Paul, R., Casati, F., Benatallah, B.: Event correlation for process discovery from web service interaction logs. The VLDB Journal 20(3), 417–444 (Jun 2011)
14. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. big'web services: making the right architectural decision. In: Proceedings of the 17th international conference on World Wide Web. pp. 805–814. ACM (2008)
15. Richardson, L., Ruby, S.: RESTful Web Services (2007)