A Temporal Data-Mining Approach for Discovering End-to-End Transaction Flows

Ting Wang[2], Chang-shing Perng[1], Tao Tao[1], Chungqiang Tang[1], Edward So[1], Chun Zhang[1], Rong Chang[1], and Ling Liu[2]

[1]IBM T.J. Watson Research Center
[2]Georgia Institute of Technology

Abstract

Effective management of Web Services systems relies on accurate understanding of end-to-end transaction flows, which may change over time as the service composition evolves. This work takes a data mining approach to automatically recovering end-to-end transaction flows from (potentially obscure) monitoring events produced by monitoring tools. We classify the caller-callee relationships among monitoring events into three categories (identity, direct-invoke, and cascaded-invoke), and propose unsupervised learning algorithms to generate rules for each type of relationship. The key idea is to leverage the temporal information available in the monitoring data and extract patterns that have statistical significance. By piecing together the caller-callee relationships at each step along the invocation path, we can recover the end-to-end flow for every executed transaction. Experiments demonstrate that our algorithms outperform human experts in terms of solution quality, scale well with the data size, and are robust against noises in monitoring data.

1. Introduction

Powered by the Web Services technology, Service-Oriented Architecture (SOA) is quickly becoming a standard way of building agile business applications to support today's highly dynamic business activities. With the SOA approach, software is packaged into reusable, coarse-grained *services* that provide certain functions through a standard Web Services interface. As business processes evolve over time, these basic services then can be quickly composed into new solutions that satisfy new business needs with minimal development time and cost.

The agility of SOA is a key reason for its success, which unfortunately also brings along with it a set of new challenges. As the composition of business applications change more often, it becomes harder to obtain an accurate image of how a transaction flows through the IT system. Architecture diagrams are often used to help understand a complex IT system, but they may lack of details or quickly become outdated as the service composition evolves.

We have repeatedly witnessed that major outages actually happened due to the lack of understanding of the *actual* dependency between business processes and the underlying IT resources, e.g., mistakenly bringing down a server for maintenance while a revenue-generating business application running half way across the globe critically depends on the server.

The purpose of this work is to understand *precisely* how one transaction instance triggered by a business activity *actually* flows through the IT system. For example, when a workflow engine initiate a process to book a flight for a customer, we want to know precisely which servers are involved in this transaction instance. This knowledge of dynamic transaction flow not only helps avoid the mistakes mentioned above and alike, but also has much broader usage: (1) discover discrepancy between architecture diagrams and the real system, (2) understand IT resource consumption and provide a basis for price charging and resource provisioning, (3) pinpoint the root cause of performance issues (e.g., which server on the invocation chain is slow and responsible for the long response time), and (4) discover the most frequent execution path across servers, which is the primary target for performance optimization (as opposed to optimization purely based on architecture diagram knowledge).

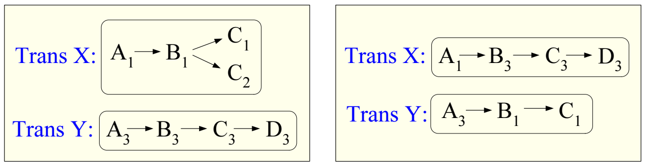
Ideally, discovering dynamic transaction flows across servers can be done automatically by monitoring tools. Unfortunately, our experience with real deployed systems and commercial monitoring tools indicate that, although this can be done for new applications developed from scratch, this is often not the case in existing deployed systems due to various reasons. For example, the servers, middleware, or applications may come from different vendors and their monitoring tools do not cooperate. It is also not uncommon that some monitoring tools simply do not have this capability.

The problem would have been easier, if every transaction instance carries a unique token when passing

[1]This work was done during Ting Wang's internship at IBM Research.

node	server name	cell
cluster name	hostname	ip address
parent time	current sequence	thread id
elapsed time	WSDL port	operation
one-way flag	current thread id	msg length
current-ticket	current time stamp	fault-flag
time stamp	parent thread id	event type
parent ticket	parent sequence	remote ip

Table 1. Sample attributes in the ITCAM log.


Trans X: $A_1 \rightarrow B_1 \rightarrow C_1$, C_2
Trans Y: $A_3 \rightarrow B_3 \rightarrow C_3 \rightarrow D_3$
Trans X: $A_1 \rightarrow B_3 \rightarrow C_3 \rightarrow D_3$
Trans Y: $A_3 \rightarrow B_1 \rightarrow C_1$

(a) Correct Transaction Flows (b) Incorrect Transaction Flows

Figure 1. Challenges in discovering individual transaction flows. (b) is an incorrect interpretation of what happens in (a).

through different servers, and the monitoring tool on every server logs the token along with the observed transaction activity. This can be done by enforcing an industry standard of event log format and semantics on every monitoring tool, which unfortunately is unlikely to happen any time soon.

On the other hand, we observe that sometimes an expert may be able to manually recover the end-to-end transaction flow by leveraging obscure information in logs generated by monitoring tools. For example, Table 1 lists some attributes in a log file. By studying a large amount of log data and matching the values of the attributes, the expert might guess the relationship between the attributes, e.g., [parent ticket ⇔ current ticket], [parent sequence ⇔ current sequence], and [parent thread ⇔ current thread]. This example uses intuitive attribute names for ease of understanding, but real monitoring data can be much more obscure. Using this knowledge, the expert may guess that server A invoked server B at a particular time, which caused B to further invoke server C right after that.

The manual process above is time consuming and error-prone as the expert guesses the syntax and semantics of log data through trial and error. We propose a data-mining approach to systematically address this challenge. We design unsupervised learning algorithms to efficiently analyze a large amount of log data, and identify a minimum set of attributes in log files that link a caller activity to a callee activity. The key idea is to leverage the temporal information available in the monitoring data and extract patterns that have statistical significance. By piecing together the caller-callee relationship at each step along the invocation path, we can recover the end-to-end flow for each transaction.

Our main contribution is not only the actual algorithms we developed, but more importantly the data-driven approach for systematically analyzing Web Services invocation logs. We believe this approach has many applications beyond the focus of this paper.

The remainder of the paper is organized as follows. Section 2 provides an overview of our system, which consists of an offline subsystem that extracts knowledge from historical monitoring data, and an online subsystem that leverages the knowledge to construct end-to-end flows for individual transactions in realtime. Section 3 presents our knowledge extraction algorithms in detail. Section 4 evaluates our proposed algorithms. Related work is discussed in Section 5. Section 6 concludes the paper.

2. System Overview

In this section, we first describe the problem of end-to-end transaction flow discovery, and then present an overview of our solution.

2.1 Problem Description

Transaction Flow. Figure 1(a) shows the flows of two transactions X and Y, which involve four Web Services modules (A, B, C, and D) running on different servers. We use subscripts to denote different invocations to the same module. For example, A_1 and A_3 are two different invocations to module A. Figure 1(b) is an incorrect understanding of the transaction flows in Figure 1(a). Specifically, B_1 is invoked by A_1, but Figure 1(b) relates B_1 to A_3. Moreover, C_2 is invoked by B_1, but Figure 1(b) relates C_2 to neither X nor Y. This example demonstrates the challenges in recovering transaction flows from concurrent transaction activities. Our unsupervised learning algorithms attempts to avoid mistakes as those in Figure 1(b) by automatically extracting knowledge from monitoring data.

Event Relationship. A monitoring tool generates a monitoring even in the log file to record an observed software activity. Figure 2 shows the events recorded

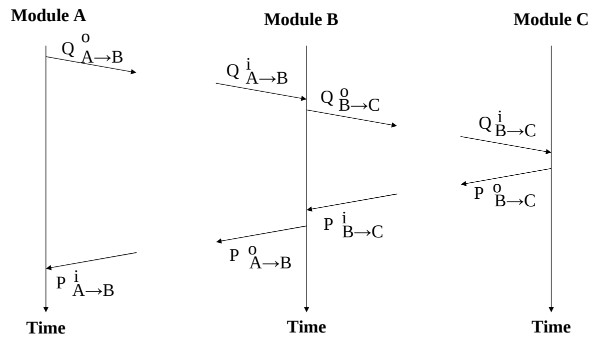


Figure 2. Events logged during a transaction with flow $A \rightarrow B \rightarrow C$.

for the transaction flow $A \rightarrow B \rightarrow C$. We only study synchronous Web Service invocation in this paper. Based our study of commercial monitoring tools, we assume that four events are logged for each Web Service invocation. For the invocation $A \rightarrow B$, the caller side records the outgoing request $Q^o_{A\rightarrow B}$ and the incoming reply $P^i_{A\rightarrow B}$. The callee side records the incoming request $Q^i_{A\rightarrow B}$ and the outgoing reply $P^o_{A\rightarrow B}$. Here Q, P, o, and i stand for re**Q**uest, re**P**ly, **O**utgoing, and **I**ncoming, respectively.

The events logged for the same transaction flow are related to one other by three kinds of relationships: *identity*, *direct-invoke*, and *cascaded-invoke*. The *identity relationship* ($\simeq$) correlates events logged at the same module for the same invocation. For example, $\left\{Q^o_{A\rightarrow B} \simeq P^i_{A\rightarrow B}\right\}$ has this relationship at the caller side, and $\{Q^i_{A\rightarrow B} \simeq P^o_{A\rightarrow B}\}$ has this relationship at the callee side. The *direct-invoke* ($\Rightarrow$) relationship links a caller identity to a callee identity, both of which are for the same invocation. For instance, $A \Rightarrow B$ fully captures the relationship between four events:

$$\left\{Q^o_{A\rightarrow B} \simeq P^i_{A\rightarrow B}\right\} \Rightarrow \left\{Q^i_{A\rightarrow B} \simeq P^o_{A\rightarrow B}\right\}.$$

Finally, the *cascaded-invoke* ($\rightsquigarrow$) relationship captures a cascaded invocation. Suppose A calls B, which further calls C. This can be represented as

$$(A \Rightarrow B) \ \rightsquigarrow \ (B \Rightarrow C) \ .$$

In our description, we use *invoke relationship* to generally refer to both the direct-invoke relationship and the cascaded-invoke relationship.

Event Representation Monitoring tools usually record events in unstructured text. We convert unstructured text into a normalized format, where each event is represented as a relational tuple:

$$\mathcal{A} = \langle \mathcal{A}^o, \mathcal{A}^t, \mathcal{A}^d \rangle \ .$$

Here $\mathcal{A}^o$ is a set of attributes specifying the operation of the event (e.g., Web Service end point), $\mathcal{A}^t$ is a set of temporal attributes (e.g., event timestamp), and $\mathcal{A}^d$ is a set of description attributes. Table 1 lists some examples of even attributes. Events produced by the same software module usually use the same set of attributes, whereas events produced by different software modules may use different sets of attributes.

Rules for Discovering Event Relationships. If a tool can accurately infer the three types of relationships between events (i.e., identity, direct-invoke, and cascaded-invoke), then it can piece together the invocation steps along the transaction path to recover the end-to-end transaction flow. A human expert guesses the relationship between two events by correlating their attribute values. Take the event attributes in Table 1 as example. After analyzing a large amount of monitoring data, the expert might guess that two events x and y are related by the invoke relationship if and only if their attribute values match as follows:

$$\begin{aligned} x \,.\, \mathit{current\_thread} &= y \,.\, \mathit{parent\_thread} \\ x \,.\, \mathit{current\_tickets} &= y \,.\, \mathit{parent\_tickets} \\ x \,.\, \mathit{current\_sequence} &= y \,.\, \mathit{parent\_sequence} \\ x \,.\, \mathit{ip\_address} &= y \,.\, \mathit{remote\_ip}. \end{aligned}$$

Given the obscurity of log file contents and the diversity of monitoring tools, it is extremely time consuming and error-prone for a human expert to manually derive and verify these relationship discovery rules. For example, after analyzing a small amount of data, the expert may conclude that the *current_tickets* attribute alone can uniquely pair up events with the invoke relationship. This rule might be true for the small amount of data the expert viewed manually, but does not hold in general for a vast amount of data the expert never viewed. Moreover, noises in the monitoring data (e.g., the log contains the requests for some invocations but not the replies) make it more difficult for the expert to make a statistically correct decision. In response to this challenge, we design unsupervised learning algorithms that automatically analyze a large amount of monitoring data and derive rules that have statistical significance.

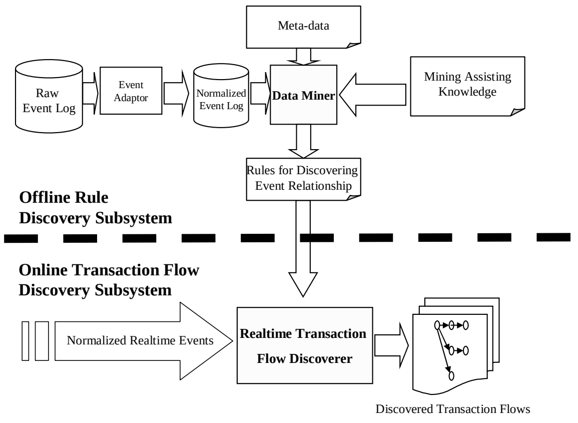

Meta-data
Raw Event Log
Event Adaptor
Normalized Event Log
Data Miner
Mining Assisting Knowledge
Rules for Discovering Event Relationship
Offline Rule Discovery Subsystem
Online Transaction Flow Discovery Subsystem
Normalized Realtime Events
Realtime Transaction Flow Discoverer
Discovered Transaction Flows

Figure 3. Simplified architecture of our system for transaction flow discovery.

2.2. System Architecture

Figure 3 show the architecture of our system. The subsystem above the dashed line runs in offline mode. It analyzes historical log data (i.e., unlabeled training data) to synthesize rules that can be used for detecting the relationships between events (i.e., identity, direct-invoke, and cascade-invoke). The subsystem below the dashed line runs in online mode. Guided by the relationship discovery rules generated by the offline subsystem, the online subsystem takes realtime monitoring events as input and constructs the end-to-end flow for every executed transaction.

The offline subsystem works as follows. First, historical raw monitoring data are parsed and normalized by the *event adapters*. In addition to the *normalized event log*, the *data miner* takes two extra inputs. The *meta data* contain information about the classification of event attributes into the operational, temporal, and descriptive categories. The *mining-assisting knowledge* is optional. It is needed only if some critical domain-specific knowledge is not available in the log data. The outputs of the data miner are rules for discovering event relationships, which drive the operation of the online subsystem.

In the online subsystem, raw monitoring events are gathered in realtime from a distributed system, normalized and then forwarded to the *realtime transaction flow discover*. The discoverer interprets rules produced by the data miner to identify events with a certain relationship. It pieces together events recorded at each step along the invocation path of a transaction, and finally generates a complete end-to-end transaction flow. These realtime transaction flows then can be utilized by the downstream tools, for example, to build a realtime monitoring dashboard for visualizing system dynamics from both the IT perspective and the business perspective.

3 Data Mining Algorithm

The *data miner* in Figure 3 takes historical monitoring data as input, performs deep analysis on the data, and then synthesizes rules that can be used for detecting the relationships between events. This section presents the data miner in detail.

The data miner takes an unsupervised approach for learning, i.e., it does not require experts to manually label a set of monitoring data as training samples, which is often a time consuming and error prone process. On the other hand, the data miner can also incorporate expert's domain knowledge to improve the quality of the generated rules.

To reconstruct end-to-end transcation flows from monitoring events, one needs to infer three types of relationships between events: *identity*, *direct-invoke*, and *cascaded-invoke* (see Section 2.1 for examples of these relationships). The data miner uses one algorithm to generate rules for discovering the identity relationship, and uses another algorithm to generate rules for both the direct-invoke relationship and the cascaded-invoke relationship, because of their similarity (i.e., events are nested in time).

Our algorithms search for the minimum set of event attributes that can uniquely identify two events with a given relationship. The general method is to (1) use events with statistical significance as learning samples, (2) establish an ideal solution that can perfectly derive event relationships, and (3) search for a practical solution that has the smallest discrepency with respect to the ideal solution. Below, we present our algorithms in detail.

Algorithm 1: Algorithm for deriving the identity attribute set $\mathcal{A}^p$ that can be used for discovering the identity relationship.

```
Input: collection of event instances $\mathcal{C}$, threshold $\delta$i.
Output: The pair attribute set $\mathcal{A}^p$.
// $\mathcal{H}$: stack of added attributes
$\mathcal{H} \leftarrow \emptyset$, $\mathcal{A}_{open} = \mathcal{A}^d$;
// pre-screen all non-promising attributes
for each attribute $a \in \mathcal{A}_{open}$ do
    if $\mathsf{Q}(\mathcal{A}^o \cup \{a\}) < \mathsf{Q}(\mathcal{A}^o)$ then $\mathcal{A}_{open} \leftarrow \mathcal{A}_{open} \setminus \{a\}$;
// incremental search for optimal combination
while $\mathcal{A}_{open} \neq \mathcal{H}$ and $\mathsf{Q}(\mathcal{A}^o \cup \mathcal{H}) < \delta$ do
    $a^* = \arg\max_a \mathsf{Q}(\mathcal{A}^o \cup \mathcal{H} \cup \{a\})$ for $a \in \mathcal{A}_{open} \setminus \mathcal{H}$;
    if $\mathsf{Q}(\mathcal{A}^o \cup \mathcal{H} \cup \{a^*\}) < \mathsf{Q}(\mathcal{A}^o \cup \mathcal{H})$ then
        // backward search
        repeat
            $a' \leftarrow$ pop top entry of $\mathcal{H}$;
            $a^+ = \arg\max_a \mathsf{Q}(\mathcal{A}^o \cup \mathcal{H} \cup \{a\})$
                for $a \in \{a | \mathsf{Q}(\mathcal{A}^o \cup \mathcal{H} \cup \{a\}) < \mathsf{Q}(\mathcal{A}^o \cup \mathcal{H} \cup \{a'\})\}$;
        until $\mathcal{H} = \emptyset$ or $\mathsf{Q}(\mathcal{A}^o \cup \mathcal{H} \cup \{a^+\}) \geq \mathsf{Q}(\mathcal{A}^o \cup \mathcal{H})$ ;
        if $\mathcal{H} \neq \emptyset$ then
            push $a^+$ to $\mathcal{H}$;
        else report failure and break;
    else
        if $\mathsf{Q}(\mathcal{A}^o \cup \mathcal{H} \cup \{a^*\}) \geq \delta$ then
            add $\mathcal{A}^o \cup \mathcal{H} \cup \{a^*\}$ to $\mathcal{A}^p$ and break;
        else push $a^*$ to $\mathcal{H}$;
```

3.1. Identity Relationship

Recall that an event is represented as a relational tuple, $\mathcal{A}=\langle \mathcal{A}^o, \mathcal{A}^t, \mathcal{A}^d \rangle$, where $\mathcal{A}^o$ $\mathcal{A}^t$, and $\mathcal{A}^d$ are operation, temporal, and description attributes, respectively. The *identity attribute set* $\mathcal{A}^p$ is a *minimum* subset of $\mathcal{A}$, such that for two events e_1 and e_2 with the identity relationship, $\mathcal{A}^p(e_1) = \mathcal{A}^p(e_2)$, whereas for two events e_1' and e_2' without the identity relationship, $\mathcal{A}^p(e_1') \neq \mathcal{A}^p(e_2')$. That is, the values of $\mathcal{A}^p$ uniquely identify two events that have the identity relationship. Because the identity relationship is established by a Web Service invocation, we know that $\mathcal{A}^o \in \mathcal{A}^p$. If we can accurately infer the identity attributes $\mathcal{A}^p$ from historical monitoring data, then in the online subsystem that continuously processes monitoring events in realtime, we can use $\mathcal{A}^p$ to identify events with the identity relationship.

Below we describe how to learn the identity attribute set $\mathcal{A}^p$ from unlabelled historical monitoring data. The key observation is that $\mathcal{A}^p$ groups event instances into pairs. If we organize event instances into a histogram according to their projection on the attributes in $\mathcal{A}^p$, the *ideal histogram* is the one where each bucket contains exactly two instances. For example, the event instances in Figure 2 should be grouped into four pairs. $\left\{Q^o_{A\to B} = P^i_{A\to B}\right\}$, $\left\{Q^i_{A\to B} = P^o_{A\to B}\right\}$, $\left\{Q^o_{B\to C} = P^i_{B\to C}\right\}$, and $\left\{Q^i_{B\to C} = P^o_{B\to C}\right\}$.

Based on this observation, for a set of attributes $\mathcal{A}' \subseteq \mathcal{A}$, we measure its quality $\mathsf{Q}(\mathcal{A}')$ as the similarity of the ideal histogram and that projected over $\mathcal{A}'$. In our current implementation, we use Euclidean distance to measure the similarity of two histograms. Our algorithm aims to find the minimum set of attributes $\mathcal{A}'$ that has the highest quality score $\mathsf{Q}(\mathcal{A}')$.

To find $\mathcal{A}'$ efficiently, our algorithm reduces the search space significantly by leveraging two key paradigms, namely *Apriori rule*[2] and *best-first search*:

- Apriori rule. Suppose $\mathsf{Q}(\mathcal{A}') > \mathsf{Q}(\mathcal{A}'')$, where $\mathcal{A}'$ and $\mathcal{A}''$ are two sets of attributes and $\mathcal{A}' \subset \mathcal{A}''$. It is safe to stop exploring the supersets of $\mathcal{A}''$, because the quality of those supersets cannot be better than $Q(\mathcal{A}')$. Note that $\mathsf{Q}(\mathcal{A}') > \mathsf{Q}(\mathcal{A}'')$ means that $\mathcal{A}''$ already have too many buckets that contain only a single event instance. Therefore, adding more attributes to $\mathcal{A}''$ can only further deteriorates the quality.

- Best-first search. Suppose the current candidate identity attribute set is $\mathcal{A}'$. We expand $\mathcal{A}'$ by first considering the attribute h that leads to the highest score $\mathsf{Q}(\mathcal{A}' \cup \{h\})$. Let h^* be such an attribute, i.e., $h^* = \arg\max_h \mathsf{Q}(\mathcal{A}' \cup \{h\})$. If score $\mathsf{Q}(\mathcal{A}' \cup \{h^*\}) < \mathsf{Q}(\mathcal{A}')$, it is safe to stop exploring other attributes.

Algorithm 1 shows the pseudo code for generating the identity attribute set $\mathcal{A}^p$ that can be used for discovering the identity relationship. We first identify the open attribute set $\mathcal{A}_{open}$ by removing all the non-promising attributes that bring no improvement to the quality of the selected set. We then apply best-first search to find the minimum set of attributes with quality above the threshold δ. On meeting the boundary, we apply a backward search strategy.

3.2. Invoke Relationship

The rules for discovering the direct-invoke relationship and the cascaded-invoke relationship are similar, and we simply refer to both of them as the *invoke relationship.* Given a collection of events with the associated set of attributes as $\mathcal{A} = \langle \mathcal{A}^o, \mathcal{A}^t, \mathcal{A}^d \rangle$, the *invoke mapping* is an one-to-one mapping m between two subsets $\mathcal{A}^i_1$ and $\mathcal{A}^i_2$ of $\mathcal{A}^d$, such that for two pairs e_1 and e_2 with the invoke relationship, $\mathcal{A}^i_1(e_1) =^m \mathcal{A}^i_2(e_2)$ (i.e., e_1 and e_2 have identical values w.r.t. the mapping m), whereas for two e'_1 and e'_2 without the invoke relationship, $\mathcal{A}^i_1(e'_1) \neq^m \mathcal{A}^i_2(e'_2)$. Note that each of e_1, e_2, e'_1, and e'_2 is a pair of events with the identity relationship rather than an individual event. Using the

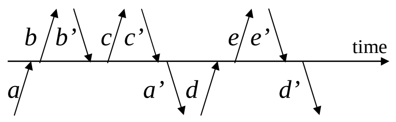


Figure 4. Rules for discovering the invoke relationship.

example in Figure 2, e_1 can be $\left\{Q^o_{A\to B} = P^i_{A\to B}\right\}$, e_2 can be $\left\{Q^i_{A\to B} = P^o_{A\to B}\right\}$. e_1 and e_2 have the invoke relationship. We use a mapping m instead of just an attribute set because the caller and callee may have different event format.

If we can accurately infer the invoke mapping m from historical monitoring data, then in the online subsystem that continuously processes monitoring events in realtime, we can use m to identify events with the invoke relationship. Below we describe how to learn m from unlabelled historical monitoring data, which is based on two key observations.

The first observation is that, if two event pairs are related by the invoke relationship, then one pair must be nested in the other in the temporal dimension. For example, in Figure 2, $\left\{Q^i_{A\to B} = P^o_{A\to B}\right\}$ is nested inside $\left\{Q^o_{A\to B} = P^i_{A\to B}\right\}$. The second observation is that two event pairs with the invoke relationshp have functional dependency on their corresponding operations. We can therefore aggregate the nested structure on the operation level, and pick statistically significant operation pairs to form the learning samples.

Concretely, our algorithm executes the following steps. (1) Group events into event pairs using the algorithm in Section 3.2, e.g., the event pairs in Figure 4: (a, a'), (b, b'), (c, c'), (d, d'), (e, e'). (2) Find nested event pairs, e.g., (b, b') is nested in (a, a'), (c, c') is nested in (a, a'), and (e, e') is nested in (d, d'). (3) Aggregate such nested structures on the operation level, e.g., (a, a'), (b, b'), (c, c'), (d, d') and (e, e') are of operations A, B, C, D, E, respectively. Therefore the counters of $A{\to}B$, $A{\to}C$, and $D{\to}E$, are each increased by one. Given the statistics of the nested relationship between operations, we pick the operation pairs with the largest counters, and use their instances as learning samples. In our example, we may pick $A{\to}B$, $A{\to}C$, and $D{\to}E$, as the correlated operation pairs, and their instances $(a, a') - (b, b')$, $(a, a') - (c, c')$, $(d, d') - (e, e')$ as the learning samples. Among the learning samples, we pick all the request instances (e.g., $a{\to}b$, $a{\to}c$, $d{\to}e$) and consider them as pairs with the invoke relationship. These pairs form our learning samples.

Our goal is to find an optimal mapping m between two subsets of $\mathcal{A}$, such that w.r.t. m (i) event pairs with the invoke relationship have identical values, and (ii) event pairs without the invoke relationship have distinct values. For the example in Figure 4, the optimal mapping should have agreed values for $a{\to}b$, $a{\to}c$, and $d{\to}e$, but different values for other pairs, e.g., $a{\to}e$.

Therefore we first find the set of attribute pairs satisfy condition (i) above as the candidate attributes, from which we then search for the *minimum* set of attribute pairs meeting condition (ii) to form the mapping m. It is straightforward to check the quality of the selected attribute pairs w.r.t. condition (i) by comparing attribute values. For condition (ii), we adopt a search strategy similar to Algorithm 1.

Specifically, our algorithm leverages the following observation. Suppose m is a mapping between two attribute sets $\mathcal{A}_1^i$ and $\mathcal{A}_2^i$, which are on the caller side and the callee side, respectively. Assuming this mapping satisfies condition (i), if condition (ii) is also satisfied, then $\mathcal{A}_1^i$ should distinguish all the distinct event instances at the caller side. However the event instances at the callee side may not have distinct values w.r.t. $\mathcal{A}_2^i$. For example, in Figure 4, a and d should have distinct values w.r.t. $\mathcal{A}_1^i$, but b and c tend to have the same values w.r.t. $\mathcal{A}_2^i$, since they are both caused by a.

Therefore, we only need to examine the instances at the caller side in search of the optimal mapping that satisfies condition (ii). Concretely, we execute the following steps. (1) From the pool of event pairs with the invoke relationship, pick all the instances at the caller side. (2) Search for the *minimum* mapping m ($\mathcal{A}_1^i \to \mathcal{A}_2^i$) such that $\mathcal{A}_1^i$ distinguishes all these instances (i.e., resulting in a histogram where the size of every bucket is one). The search algorithm is similar to Algorithm 1, and is omitted here.

4 Evaluation

This section presents an empirical evaluation of our algorithms that generate rules for discovering event relationships (i.e., identity, direct-invoke, and cascaded-invoke). We use $\mathcal{ID}$ to denote the rules for discovering the identity relationship. We use $\mathcal{IV}$ to denote the rules for discovering both the direct-invoke relationship and the cascaded relationship as they are similar.

We use three metrics to evaluate our algorithm. (1) Is the quality of the rules generated by our algorithm comparable to those found by human experts? (2) How sensitive are our algorithm to noise in the monitoring data? (3) Are our algorithm scalable in terms of execution time as the data size increases? In summary, our experiments show that our algorithm generates high-quality rules, is robust against noises, and is scalable.

The experiments are based on the travel booking application shown in Figure 5. The *TravelBookingService* is hosted in a workflow engine (WebSphere Process Server 6.1). It contains one business activity *BookFlight*, which internally invokes a series of Web Services (*BookFlight, EPayment, CreditCard*, and *EMail*) running on different servers. All the servers are hosted in a IBM BladeCenter and run Windows Server 2003. The entire distributed system is monitored by the ITCAM4SOA component of IBM Tivoli Monitoring (ITM) 6.1. Some attributes of the events logged by the monitoring tool are shown in Table 1.

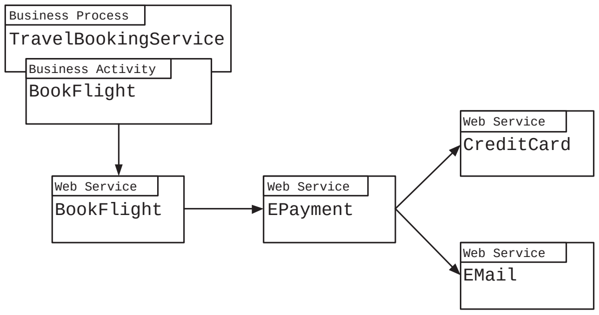


Figure 5. The travel booking application.

4.1 Quality

We first evaluate the quality of the relationship discovery rules generated by our algorithm by comparing them with rules manually generated by human experts. Before developing the automated data miner described in this paper, we actually have already used those manual rules in production systems for a long period of time. Therefore, those manual rules are not just designed for evaluation purpose and are a fair baseline for comparison. The reason that motivated this work is our observation that it is extremely time consuming and error prone for experts to manually study a large amount of logging data in order to derive the rules. With the automated data miner, we actually can verify that the manual rules sometimes are inaccurate even if the domain experts have spent extraordinary efforts in constructing them.

Table 2 summarizes and compares the rules, with the differences highlighted in bold. We make the following observations. (1) Our algorithm automatically discovers that the attributes WSDL port and operation are redundant in identifying the identity relationship, because either of them, in combination with the other attributes, can uniquely identify events with the identity relationship. This is empirically proved by the fact that the rules suggested by human expert and our algorithm achieve the same quality score of 0.948, as defined in Section 3.2. (2) Our algorithm discovers that it is not necessary to use the attribute pair *Current_Ticket*→*Parent_Ticket* for detecting the identity relationship, because the other three attributes are already sufficient. This is empirically proved by the fact that the rules discovered by human expert and our algorithm achieve the same quality score of 1.0.

Because of the redundancy in rules, the expert rules are not optimal compared with the rules derived by our algorithm. This experiment demonstrates that the data miner can generate relationship discovery rules comparable to, or even better than, the expert rules. On the other hand, we acknowledge that, sometimes some application domain knowledge is not expressed in the monitoring data in anyway, and hence it is impossible for the data miner to discover it. In this case, it

rule category		specification
$\mathcal{ID}$	experts	current thread, current sequence, ip address, **WSDL port, operation**
	algorithm	current thread, current sequence, ip address, **WSDL port/operation**
$\mathcal{IV}$	experts	current thread $\rightarrow$ parent thread **current ticket $\rightarrow$ parent ticket** current sequence $\rightarrow$ parent sequence ip address $\rightarrow$ remote ip
	algorithm	current thread $\rightarrow$ parent thread current sequence $\rightarrow$ parent sequence ip address $\rightarrow$ remote ip

Table 2. Summary of correlation rules discovered by human expert and our algorithm.

is beneficial to have experts provide that knowledge to our data miner as extra input.

4.2 Robustness

In real systems, the monitoring data may be incomplete or contains errors. These noises make it harder for the data miner to derive the relationship discovery rules. However, we will show that the data miner works robustly even in the face of significant noises in the monitoring data.

Here we specifically consider noises caused by message loss, i.e., some invocations may not have coupled requests and replies, for example, due to communication errors or problems with the monitoring tool. The message loss impacts rule discovery by making event pairing more difficult.

We evaluate the robustness of our algorithm against message loss. We measure the quality of the discovered rules under various message loss rates, and determine the threshold of loss rate above which the discovered rules deviate from those found under the perfectly clean data. Specifically, the thresholds for $\mathcal{ID}$ and $\mathcal{IV}$ are:

$$\mathcal{ID} : 56\% \qquad \mathcal{IV} : 93\% \ .$$

We make several observations. (1) The quality of the relationship discovery rules start to deteriorate only if the message loss rate is extremely high (56% for $\mathcal{ID}$ and 93% for $\mathcal{IV}$). Note that 93% means 93% of the invocation messages (request or response or both) are not recorded in the log data. This indicates that our algorithm is extremely robust against noises that may exist in real environments. (2) The $\mathcal{IV}$ threshold is much higher than the $\mathcal{ID}$ threshold. This can be explained by the fact that $\mathcal{IV}$ is discovered by analyzing the operation pairs with the highest statistical correlation, and such statistical significance is preserved even in extremely noisy data.

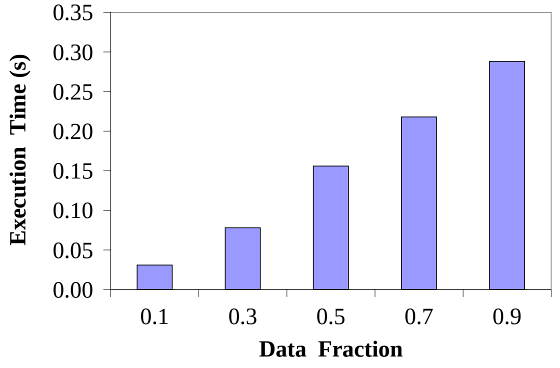


Figure 6. $\mathcal{ID}$: execution time vs. fraction of data collection.

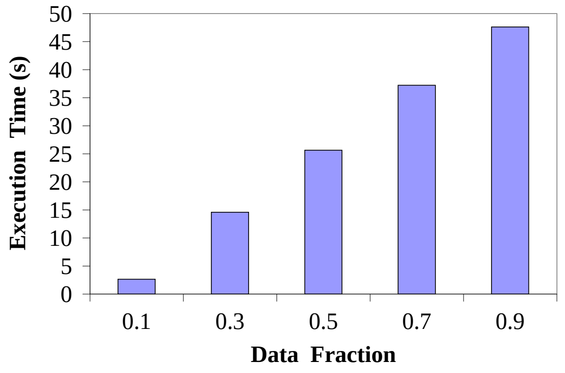


Figure 7. $\mathcal{IV}$: execution time vs. fraction of data collection.

4.3 Scalability

The last experiment evaluates the scalability of our algorithm (see Figures 6 and 7). We measure the execution time of the algorithm for a given fraction of the data collection. The execution time grows almost linearly with the data size. The $O(n)$ complexity is necessary because any algorithm that generates relationship discovery rules needs to at least scan the data once. On the other hand, because our algorithm is extremely robust against message loss, for a very large data set, we can select a subset of the data and use it as the input for our algorithm. The figures also show that generating $\mathcal{IV}$ is slower than generating $\mathcal{ID}$ by two orders of magnitude.

5. Related Work

Most of the prior work leverages ontological or semantic clues to finding component-level static relationship. By contrast, it is much more challenging to discover the end-to-end flow for individual transaction instances in realtime. To our knowledge, this work is the first one that takes an unsupervised learning approach to address this challenge.

The work of De Pauw *et al.*[6] is the closest to ours. Their method explores the semantic relationship between event schemas, and iterates over pairs of paths between any two event schemas to determine whether there is a one-to-one, one-to-many, or many-to-many relationship. Their method can discover dialogs across multiple transactions, e.g., three related orders placed by the same customer, but requires access to the full contents of Web Services messages, which unfortunately is not always available, for example, due to security or privacy concerns. By contrast, our algorithms take ordinary log files as input and perform deep histogram and temporal analysis.

Kind *et al.*[9] used "shockwaves" in IP networks to discover dependency. They described a mathematical model for discovering transaction flows by correlating NetFlows at the packet level, where a NetFlow as a set of network packets during a certain time interval. Similarly, Brown *et al.*[4] actively created perturbation in the system for dependency discovery. By contrast, we take a passive monitoring approach and use information at the task level.

Transaction flow discovery has also been studied in the performance debugging area. Aguilera *et al.*[3] showed that it is possible to use low-level traces with little semantic knowledge to discover multi-hop causal path patterns. This work discovers static relationship at the component level rather than dynamic transaction flows at the instance level. Similarly, several other work [1, 7, 5] also proposed solutions for discovering component dependency in order to solve the performance debugging problem.

Unlike the methods above, Sycara [10] used Semantic Web to model Web Services invocations. This method, however, requires substantial domain knowledge about the application.

6. Conclusion

One key challenge in realtime SOA management is to understand how invocations are chained together to form end-to-end transaction flows. Much work has been done on discovering *component-level* dependency, e.g., using ontology or shockwaves. By contrast, our system can construct in *realtime* the end-to-end flow of *every* executed transaction, powered by the knowledge for identifying the relationships among monitoring events. This knowledge is extracted offline from historical monitoring data by our unsupervised learning algorithms.

We classify the temporal relationships among monitoring events into three categories. The *identity* relationship matches a Web Service invocation's request event with reply event at the same component. The *direct-invoke* relationship connects an event on the caller side to the corresponding event on the callee side, both of which are for the same invocation. The *cascaded-invoke* relationship chains two nested invocations, where A calls B and B further calls C. The end-to-end transaction flow can be recovered accurately if one can infer the three types of relationships at each step along the invocation path.

Our unsupervised learning algorithms discover the three types of relationships by searching for the minimum set of event attributes that can uniquely identify two events with a given relationship. The general method is to (1) use events with statistical significance as learning samples, (2) establish an ideal solution that can perfectly derive event relationships, and (3) search for a practical solution that has the smallest discrepancy with respect to the ideal solution. Experiments demonstrate that our algorithms outperform human experts in terms of solution quality, scale well with the data size, and are robust against noises in monitoring data.

This paper lays out a foundation for some interesting future work. Equipped with the capability of discovering end-to-end transaction flows on the instance level, one can build a realtime monitoring dashboard to visualize system dynamics from both the IT perspective and the business perspective. Data mining can also be applied to analyze historical transaction flows and identify opportunities for further optimization.

References

[1] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs and run-time behavior models. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 2004.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.

[3] M. K. Aguilera, J. C. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM Symposium on Operating Systems Principles*, October 2003.

[4] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[5] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *International Conference on Dependable Systems and Networks*, 2002.

[6] W. De Pauw, R. Hoch, and Y. Huang. Discovering conversations in web services using semantic correlation analysis. In *IEEE International Conference on Web Services*, 2007.

[7] J. Gao, G. Kar, and P. Kermani. Approaches to building self healing systems using dependency analysis. In *The IFIP/IEEE International Symposium on Integrated Network Management*, 2004.

[8] JBoss.org. Drools. http://www.jboss.org/drools/.

[9] A. Kind, D. Gantenbein, and H. Etoh. Relationship discovery with netflow to enable business-driven it management. In *1st IEEE/IFIP Int. Workshop on Business-Driven IT Management (BDIM 2006) in conjunction with NOMS'06*, April 2006.

[10] K. P. Sycara. Dynamic discovery, invocation and composition of semantic web services. In *he Third Hellenic Conference on Artificial Intelligence, SETN*, 2004.