

# Deriving Protocol Models from Imperfect Service Conversation Logs

Hamid R. Motahari-Nezhad, *Member, IEEE*,  
Régis Saint-Paul, *Member, IEEE Computer Society*,  
Boualem Benatallah, *Member, IEEE*, and Fabio Casati

**Abstract**—Understanding the *business* (interaction) protocol supported by a service is very important for both clients and service providers: It allows developers to know how to write clients that interact with a service, and it allows development tools and runtime middleware to deliver functionalities that simplifies the service development life cycle. It also greatly facilitates the monitoring, visualization, and aggregation of interaction data. This paper presents an approach for discovering protocol models from real-world service interaction logs. It presents a novel discovery algorithm, which is widely applicable, robust to different kinds of imperfections often present in real-world service logs, and able to derive protocols of small sizes targeted for human understanding. As inferring the most precise and concise model is not always possible from imperfect service logs using purely automated method, the paper presents a novel method for user-driven refinement of the discovered protocol models. The proposed approach has been implemented and experimental results show its viability on both synthetic and real-world data sets.

**Index Terms**—Web services, business protocols, noise handling in service logs, protocol discovery, protocol refinement.

## 1 INTRODUCTION

A trend that is gathering momentum in Web services is to include as part of the service description not only the service interface (WSDL [1]) but also the *business protocol* supported by the service. A business protocol is the specification of all possible conversations that a service can have with its partners [1]. A *conversation* consists of a sequence of messages exchanged between two or more services to achieve a certain goal, e.g., to order and pay for goods.

Modeling business protocols brings several benefits to Web services:

1. It provides developers with information on how to program clients that can correctly interact with a service.
2. It allows the middleware to verify that conversations are carried on in accordance with the protocol specifications, thereby relieving developers from implementing the exception handling logic.
3. It allows the middleware to check if a service is compatible (can interact) with another or if it

conforms to a certain standard, thereby supporting both service development and binding [2].

4. It provides the basis for monitoring and analyzing conversations, as the availability of a model can greatly facilitate the exploration and visualization of *conversation logs* (logs storing messages exchanged among services).

In Web services, standard languages and tools are also being developed to allow for specification of service interaction models (e.g., WS-BPEL and WS-CDL [1]).

This paper investigates the problem of discovering protocol models by analyzing real-world service conversation logs [3], [4]. There are several reasons why protocol discovery is needed:

1. *Protocol definition.* In real-world settings, the protocol definition may not be available. This can happen for various reasons: for example, the service has been developed using a bottom-up approach by simply SOAP-ifying a legacy application for which the protocol specification was not available [1]. Even when the protocol information is documented, the protocol specification has to be extracted from the textual description, which is not as precise as a formal model.
2. *Protocol verification, conformance checking, and evolution.* Protocol discovery is useful to verify if the designed protocol is faithfully followed in the service interactions. Similarly, conformance of service implementation with specifications issued by standardization body or industry consortium can be checked. Finally, as the implementation of service evolves, the protocol definitions become incorrect. Automated protocol discovery helps in maintaining a correct and up-to-date protocol definition.

• H.R. Motahari-Nezhad and B. Benatallah are with the School of Computer Science and Engineering, The University of New South Wales, K17 (CSE), NSW 2052, Sydney, Australia.

E-mail: {hamidm, boualem}@cse.unsw.edu.au.

• R. Saint-Paul is with CREATE-NET, Via alla Cascata 56/D Povo, 38100 Trento, Italy. E-mail: regis.saint-paul@create-net.org.

• F. Casati is with the Department of Information and Communication Technologies, University of Trento, Sommarive st. 14, Povo, 38100 Trento, Italy. E-mail: fabio.casati@dit.unitn.it.

Manuscript received 5 June 2007; revised 17 Mar. 2008; accepted 21 Apr. 2008; published online 30 Apr. 2008.

For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2007-06-0256.

Digital Object Identifier no. 10.1109/TKDE.2008.87.

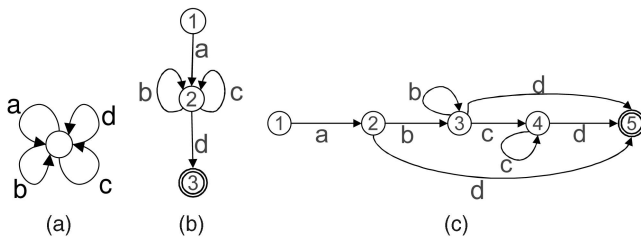


Fig. 1. (a) Flower model protocol for messages  $a$ ,  $b$ ,  $c$ , and  $d$ . (b) FSM to be discovered. (c) The same FSM in absence of some conversations.

Following our previous work [2], we adopt finite state machines (FSM) as a formalism for protocols, where states denote the possible stages a service may go through during a conversation, and where transitions are associated with messages with the meaning that in a given state only messages coupled to outgoing transition are accepted by the service (e.g., see Fig. 3).

### 1.1 In Search of a Protocol Discovery Problem Statement

Ideally, if we have a *perfect* log, i.e., a log that includes all possible conversations modulo loops (*complete log*) and that is not affected by noise (*noiseless log*, i.e., a log in which the conversations as logged are in fact those that occurred), the discovery problem could be stated as follows: *Find the model of smallest size that can accept all and only the conversations in the log.* Given two models that can describe the same log, the smallest one is generally preferable as it is easier to understand for users. For example, if state machines are used, the smallest model is the one with the fewest number of states. However, identifying the model that fulfills above requirements is challenging. Consider the extreme case of the smallest possible model (only one state), illustrated in Fig. 1a for messages  $a$ ,  $b$ ,  $c$ , and  $d$ . This model is *overgeneralized* because it accepts any conversations consisting of  $a$ ,  $b$ ,  $c$ , and  $d$  while only a subset of the possible conversations formed from these messages may be actually present in the log. At the other extreme, it is possible to build a state machine with a path for each distinct conversation of the log (also referred to as *prefix tree*). Such a model would be difficult to understand, having too many states. Moreover, it is too precise since it accepts *only* the conversations in the log. It has been shown that finding the minimal model fulfilling the above requirements is undecidable [5]. Nevertheless, algorithmic and probabilistic approaches for finding an approximate model exist [6].

However, the problem formulated as such is unrealistic for real-world service logs. As widely observed [7], [8], [9], [10], real-world logs are imperfect, i.e., they are both incomplete (correspond to a subset of possible execution) and noisy (e.g., do not record some messages). Experiments with commercial logging tools, e.g., HP SOA Manager (available at [managementsoftware.hp.com/products/soa/](http://managementsoftware.hp.com/products/soa/)), also confirm that noise is introduced in the logging process. The problem is that even a small level of imperfection (*small* noise level or *quasi* complete logs) causes an explosion in the number of states of an inferred model. Errors will lead to creating new paths in the model, which do not exist in the real protocol. Hence, models

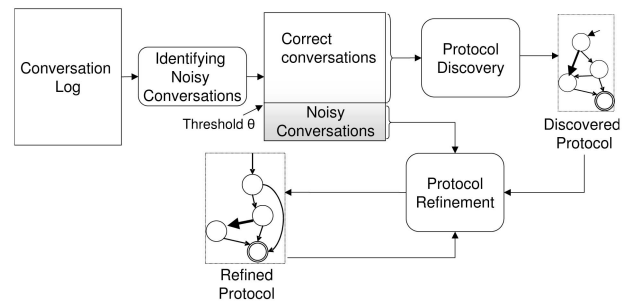


Fig. 2. Stages of proposed protocol discovery approach.

obtained from an imperfect log are not only inaccurate, they may also be too complex to be useful.

The issue for incompleteness is more subtle but equally problematic. Consider the service protocol depicted in Fig. 1b. Conversations allowed by this protocol include  $ad$ ,  $abcd$ ,  $abd$ ,  $acbd$ , and  $acd$ . If the log does not contain conversations such as  $acbd$  (and generally  $ac + b + d$ ), an automata inference algorithm [6] (which does not handle incompleteness) may infer the protocol depicted in Fig. 1c. This protocol conforms to the conversations found in the log, i.e., it correctly accepts all the logged conversations and rejects the conversation  $acbd$ , but it is more complex than the real protocol.

The above observations show that proposing a solution for discovering protocols from imperfect logs is a challenging issue. An algorithm that guarantees minimality cannot be devised, while precision can be at odds with complexity due to log incompleteness, and in any case, the presence of noise means that we can never be sure that the discovered model is accurate.

### 1.2 Contributions

The key contributions of this paper are algorithms, methodologies, and tools to support the discovery of protocol models from imperfect service logs. We focus on addressing the abovementioned problems in a three-stage approach depicted in Fig. 2.

#### 1.2.1 Identifying Noisy Conversations

Noise in conversations is usually infrequent and random. A known approach to deal with noise in logs is to use a frequency threshold to filter noisy data [11], [12], [13].

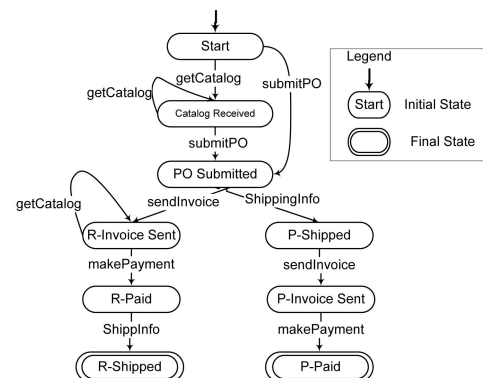


Fig. 3. The business protocol of the Retailer service.

However, the challenge is to determine an appropriate frequency threshold. In some contexts (e.g., when the error rate of the logging infrastructure is known), it is possible to rely on a manual setting of this threshold [11], [12], [13]. We propose in Section 3 a quantitative measure and an effective algorithm to determine which conversations are noisy by automatically computing a noise threshold.

In a nutshell, we proceed by analyzing the frequency distribution of unique conversations (more accurately, *subsequences* of conversations). Since noise is random, it creates new message sequences, often different from those usually produced by the service. This means that many of noisy sequences will have the same and very low frequency. In contrast, the correct sequences have varied and higher frequencies. We rely on this difference in terms of distribution to decide on a frequency threshold. Using subsequences of conversations has the advantage that they are more robust to noise than conversations (less likely to be noisy). In addition, a noisy conversation may have several correct subsequences. Thus, even noisy conversations can contribute to the identification of correct sequences.

### 1.2.2 Protocol Discovery

We propose a robust algorithm that aims at discovering simple (small sized) yet precise protocol models in the presence of log imperfections. The proposed algorithm first constructs an overgeneralized simple model of the protocol by analyzing message sequences in the log. Then, it progressively refines the initial model to ensure that conversations considered noisy or not present in the log are not accepted, thereby improving the precision of the model (See Section 4).

The proposed algorithm also caters for log incompleteness by using a number of heuristics to predict missing conversations. These heuristics exploit domain knowledge and statistical properties of the conversations in the log to infer that certain conversations are missing. For example, if through statistical analysis it becomes apparent that a certain operation can be invoked at any time, this operation will be represented as a self-transition (i.e., transition to the same state), avoiding the creation of superfluous states. Doing so, our approach allows discovering simpler models, which are also closer to the underlying protocol. To the best of our knowledge, no existing model discovery approach considers the issue of log incompleteness.

### 1.2.3 Protocol Refinement

Log imperfections make it impossible, in practice, to derive the correct protocol model by an automated approach. We propose a novel approach to *visual, interactive, and user-driven protocol refinement* (Section 5) to compensate for errors made during discovery step (Fig. 2): 1) we augment discovered protocol with various metadata including (final) state and transition supports to represent uncertainty due to log imperfection. This refinement helps to improve the precision of the discovered protocols via excluding conversations that should not have been accepted. Moreover, to compensate for the exclusion of correct but infrequent conversations, 2) we present an approach for analyzing conversations that are in the log but are not accepted by the discovered model. This approach is based on defining i) a *distance* between the protocol model and excluded conversations

and ii) *manipulations* that allow correcting the model. We show that proceeding by manipulation operations instead of by conversations facilitates the model refinement. This refinement helps in improving the recall of the discovered model, i.e., the percentage of correct conversations that are accepted by the model.

### 1.2.4 Protocol Discovery Tool

The final contribution of this paper is the implementation of the proposed approach in a tool and its validation via experiments performed on actual service execution logs (Section 6). The tool is part of ServiceMosaic project [4], a platform for model-driven analysis and management of Web service protocols.

The paper is organized as follows: In Section 2, we give basic definitions and characterize imperfections in service logs. Section 3 presents our approach for identifying noisy conversations. In Section 4, we present the protocol discovery algorithm. Section 5 explains the proposed protocol refinement approach. Implementation and evaluation of the proposed approaches are discussed in Section 6. Related work is presented in Section 7. We conclude and introduce future work in Section 8.

## 2 PRELIMINARIES AND ASSUMPTIONS

### 2.1 Modeling Business Protocols

Following our previous work [2], we choose to model business protocols as deterministic FSM as it is simple and suitable for modeling reactive behaviors. This choice is also justified by the deterministic nature of the service interactions in the sense that a service client expects to know which messages can be sent/received at any time during the interactions. We stress, however, that many of the concepts presented here apply regardless of the chosen protocol formalism.

**Definition 2.1.** *A business protocol is a tuple*

$$P = (S, s_0, F, M, T),$$

where  $S$  is the set of states of the protocol,  $M$  is the set of messages supported by the service,  $T \subseteq S \times S \times M$  is the set of transitions,  $s_0$  is the initial state, and  $F$  represents the finite set of final states. A transitions from state  $s$  to state  $s'$  triggered by the message  $m$  is denoted by the triplet  $(s, s', m)$ .

In FSM-based protocol models, states represent the different phases through which a service may go during its interactions with a client (i.e., during a conversation). Each state is labeled with a logical name, such as PO Submitted. A protocol has one initial state and one or more final states. Transitions are labeled with message names, with the semantics that the exchange of a message (with the conversation in a given state) causes a state transition to occur. FSM used to represent protocols are deterministic, meaning that for any given message, there is at most one corresponding transition from any given state to the next. A protocol  $P$  is said to *accept* (respectively, *reject*) the sequence of messages of a conversation  $c$  if a path (necessarily unique) exists (respectively, does not exist) from the initial state to one of the final states. It should be noted that constructs

such as parallelism and synchronization are not needed in the biparty interactions between two services, and the adopted model (FSM) does not support them.

**Example 2.1.** Consider a Retailer service that has two types of clients: *regular* and *premium*. A typical conversation of regular clients may start with a request for the product catalog, followed by an order. Then, an invoice is sent to clients, and once the invoice has been paid, the requested goods will be shipped. For premium customers, goods may be shipped immediately after placing an order (the invoice is sent later). The protocol of the Retailer service is depicted in Fig. 3. R-Shipped and P-Paid are final states (for regular and premium customers, respectively). In the following, we use shortened forms of these messages for simplicity: Cat, PO, Inv, Pay, and Ship for `getCatalog`, `submitPO`, `sendInvoice`, `makePayment`, and `ShippingInfo`, respectively.

## 2.2 Service Conversation Logs

We assume that service (message) logs contain the following information: 1) message transfer information (sender, receiver, and time stamp), 2) message name, and 3) a conversation ID, which is a way to associate messages to conversations. This assumption is consistent with the information that can be logged by commercial service monitoring tool [8], [14]. The possible exception is the conversation ID, which some tools may not log. In such a case, a preprocessing is needed to relate messages of the log to their corresponding conversation. The problem of correlating messages to conversations is outside of the scope of this paper and constitutes an independent research thread in its own right (see Section 8).

**Definition 2.2.** A message log  $ML$  is a collection of entries  $e = (cid, s, r, \tau, m)$ , where  $cid$  is the conversation identifier,  $s$  and  $r$  denote the sender and the receiver of message  $m$ , and  $\tau$  is the time stamp.

Conversation logs are obtained from message logs by grouping entries by conversation ID and ordering them by time.

**Definition 2.3.** A conversation log  $CL$  is a collection of conversations  $CL = \{c_1, c_2, \dots, c_n\}$ . Each conversation  $c_i \in CL$  is a sequence of messages  $c_i = \langle m_1^i, m_2^i, \dots, m_k^i \rangle$ . Conversations in  $CL$  are obtained from a message log by grouping entries per conversation and by ordering them according to their time stamp.

A conversation  $c$  can be seen as a sequence of symbols, where each symbol corresponds to a message name.

## 2.3 Characterizing Imperfection in Service Logs

There are mainly two types of imperfections in service logs: *incorrect conversations*, and *log incompleteness*.

### 2.3.1 Incorrect Conversations

In service logs, we found that the following types of problems are common:

*Missing messages.* The logging infrastructure may fail to record one or more messages of a conversation. For example,

for conversation  $abcde$ , we may have  $acde$  captured in the log, in which  $b$  is missing. This type of error happens for various reasons, including bugs in the logging infrastructure, performance degradation, or unexpected interruptions due to malfunctioning of the underlying software platforms (e.g., operating system).

*Swapping messages.* The order of messages as recorded in the log may differ from the real ordering of messages as exchanged between services. For example, for conversation  $abcde$ , we may find  $acbde$  recorded in the log, in which the order of  $b$ , and  $c$  is swapped. This type of error may be due to the granularity of time stamps of messages or performance degradation so both  $b$  and  $c$  get the same time stamps. Also, if conversations are logged on more than one server, it can be difficult to establish the correct ordering between messages of the same conversation [8].

*Partial conversations.* We call *partial* a conversation that is interrupted before its completion. This can be due to, e.g., network failure, client abortion, or service execution exceptions. For instance, if  $abcd$  is a complete conversation, the sequence  $abc$  represents a partial conversation. Although this problem may seem similar to the missing message problem, it impacts the protocol discovery in a different way: In protocols, states in which it is legal to terminate a conversation are marked as *final*. For example, in the protocol in Fig. 1c, state 5 is marked as final, which indicates that  $abcd$  is complete. The presence of partial conversation  $abc$  in the log could lead a protocol discovery algorithm to mark state 4 as final while it is not.

We refer to conversations logged with one or more missing or swapped messages, the first two of the above error types, as *noisy conversations*. Noisy conversations may lead to discover wrong and complex protocols.

### 2.3.2 Log Incompleteness

In practice, conversation logs are often incomplete, i.e., they do not contain all the possible conversations allowed by the service protocol. Incompleteness makes it difficult for a model discovery algorithm to discover simple models, as illustrated in the example in Fig. 1. We present an approach for handling incompleteness in service log in Section 4.2.

## 3 IDENTIFICATION OF NOISY CONVERSATIONS

### 3.1 Characterizing Noise Distribution

As discussed in Section 2.3, conversations in  $CL$  could be noisy, i.e., may include one or more errors (missing message or swapped message). In this section, we seek to identify noisy conversations. For simplicity, we will use in the following a Poisson process [15] as model of error distribution in the log. In the vocabulary of log, a Poisson process is defined by the following properties: 1) the number of errors in nonoverlapping sequences of log entries is independent for all sequences, 2) the probability of exactly one error on a given sequence is proportional to the sequence length, and 3) the probability of two or more errors in a sufficiently small sequence is essentially null.

Note that the actual error distribution in the log may not strictly follow a Poisson process. If the dependency between messages and errors is important, our approach will fail to identify the resulting conversations as noisy. Indeed, the

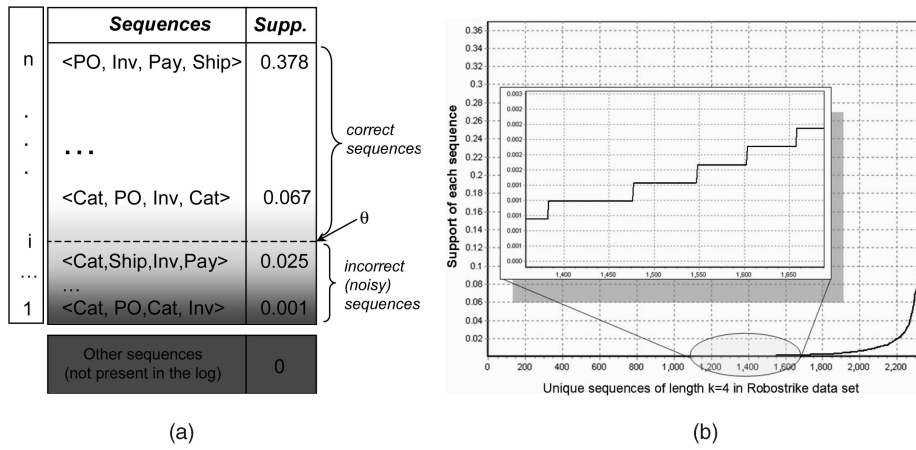


Fig. 4. (a) Support table for unique sequences of length 4 in `Retailer` log. (b) Typical distribution of the support of unique sequences in a conversation log.

more deterministic is an error, the more difficult it is to differentiate it from the normal behavior of a service. Identifying this type of near-deterministic errors in log systems is not among our claims. Such errors may be identified by applying model checking tools on the service implementation specifications [10].

### 3.2 Filtering Noisy Conversations

Based on above, a first observation is that the longer is a conversation, the more likely it contains an error. This implies that short sequences, say, of length  $k$ , are more robust to noise than long ones. Hence, we extract subsequences (of length  $k$ ) of conversations for noise analysis. An additional merit of this approach is that even very long conversations, which may be noisy, consist of several subsequences, most of them are correct. Thus, noisy conversations, too, can contribute to the identification of correct sequences. Note that our methodology does not rely on the exact distribution of errors and, therefore, is not relying on the uniformity of the density. The fact that short sequences are less likely to contain errors than long ones remains valid even under reasonable nonuniform exception density hypothesis.

For the length of the sequences, we have learned through experiments (see Section 6) that appropriate values typically range between 3 and 5. A smaller value of  $k$  implies too few distinct sequences, and this makes it hard to discriminate between noisy and correct sequences. A greater value of  $k$  increases the computational cost and tends to nullify the benefit of using sequences as opposed to conversations. We experienced that the value of  $k = 4$  works very well in most situations. This value can, however, be changed by the user according to the discovery goals and characteristics of data set in hand. This may be needed only for services that have very long conversations with high variability or very low noise. In the following, we denote by  $Q^k$  the set of *unique* sequences of length  $k$  found in the log  $CL$ . We use  $c^k$  to denote all unique sequences of a conversation  $c$ .

**Example 3.1.** Consider conversation  $c = \langle abcde \rangle$ . The sequences of length  $k = 3$ , i.e.,  $c^3 = \{\langle abc \rangle, \langle bcd \rangle, \langle cde \rangle\}$ . Consider now  $CL = \{abcd, acbd, acb\}$ . We have

$$Q^3 = \{\langle abc \rangle, \langle bcd \rangle, \langle acb \rangle, \langle cbd \rangle\}.$$

A second observation is that protocols and services in general are designed by humans, and hence, they tend to be fairly simple models. Even though a protocol may allow for a wide variety of individual conversations, the number of distinct individual sequences of small length—corresponding to portions of the protocol—should remain relatively small. However, by introducing subtle variations within correct sequences, we can expect the noise to introduce a large number of new unique sequences. Furthermore, we can expect the frequency of these new sequences to be very low, since there is no reason for random and infrequent errors to affect repetitively a same sequence in the same way.

Intuitively, this means that infrequent message sequences are likely the result of noise. The challenge lies in defining what “infrequent” means, that is, identifying a frequency threshold, denoted by  $\theta_k$ , for sequences of size  $k$ . For each distinct sequence, we compute the support, denoted by  $supp$ , as ratio of the number of conversations that contain this sequence divided by the total number of conversations. We use this measure to order the table of  $n$  unique sequences  $(q_1, q_2, \dots, q_n)$  such that  $\forall i < n, supp(q_i) \leq supp(q_{i+1})$  and  $q_i \in Q^k$  (Fig. 4a). Fig. 4b represents the histogram of sequence support for the game service conversation log (details of this data set is presented in Section 6).

This histogram can be seen as a step function. We call *step points*  $m$  the points in the ranked histogram, where the function has a step ( $supp(q_m) > supp(q_{m-1})$ ), i.e., where support value changes. We use  $l(m)$  to denote the length of the step (the number of sequences that have the same support of the previous step point). Consider now the ratio  $\gamma_m$  between the relative length of the step  $l(m)/n$  (normalized based on the total number of sequences) and the support  $supp(q_m)$  of the next step:

$$\gamma_m = \frac{l(m)}{n \cdot supp(q_m)}.$$

This value is decreasing with  $m$  since the sequences of higher support are less likely to share a same support. For some  $m_0$ ,  $l(m_0)/n$  becomes smaller than  $supp(q_{m_0})$  (i.e.,  $\gamma_{m_0} < 1$ ), and we set  $\theta_k = supp(q_{m_0})$ . Indeed, if the conversation log is large, with typical distributions of sequence frequencies, for some sequence  $m_1$  with a high

TABLE 1  
Characteristics of the Data Sets

	<i>Retailer</i>	<i>Robostrike</i>
# of operations ( $n$ )	10	32
# of conversations in $CL$	5,000	25,804
Ariety of each operation ( $\phi$ )	2.5	4.22
Noise level	9.78	?
Min. length conversation	1	1
Avg. length of conversations	9.45	43.31
Max. length conversation	35	1,921

frequency, we have a unique  $supp(q_{m1})$ , i.e.,  $l(m_1) = 1$  (excluding some rare cases, where all sequences may have the same frequency, e.g., 1234). Hence, to have  $\gamma_{m1} < 1$ , we need  $n * supp(q_{m1}) > 1$ . The value of  $supp(q_{m1})$  is between 0 and 1 (in most data sets it is below 0.5). Assuming that  $supp(q_m) = 0.1$ , we should have  $n > 10$ . Considering extreme cases, e.g.,  $supp(q_{m1}) = 0.01$ , it suffice to have  $n > 100$ . However, in most practical cases,  $supp(q_{m1})$  is typically higher than these values, and also, the number of sequences in the log is generally high (e.g., for Robostrike data set, we have  $n > 2,200$ ). This implies that, in most practical cases, we can expect that there is a point  $m$  for which  $\gamma_m < 1$ .

### 3.3 Time Complexity of the Noise Identification

The size of the histogram depends on the number of unique sequences. In a protocol involving  $p$  different operations, the number of possible unique sequences is given by  $p^k$ . This corresponds to a scenario where any operation can follow any other operation. In practice, each operation is followed by only a subset of operations. If  $\phi$  denotes the average number of operations that can follow a given operation, an estimate of the number of unique sequences of length  $k$  is given by  $p * \phi^{k-1}$ , where  $\phi \ll p$ . For example, in our experiments (see Table 1), with a protocol of 32 operations, less than 2,400 unique sequences of length 4 were observed (Fig. 4b), among 25,804 conversations in the log. This is much smaller than the  $32^4 \approx 10^6$  theoretical maximum. In this case,  $\phi \approx 4.22$  (see Table 1). Finally, the number of unique sequences is independent of the size of the data set (the number of conversations in  $CL$ ), i.e., new sequences are found less and less frequently as the data set grows. Hence, the time complexity of the noise identification depends linearly on the number of conversations (as we parse the data only once).

## 4 PROTOCOL DISCOVERY ALGORITHM

Let  $\theta_k$  be the noise thresholds for sequences of length  $k$  estimated using the approach presented in Section 3. We denote by  $Q_\theta^k = \{q \in Q^k | supp(q) \geq \theta_k\}$ , the set of correct sequences of length  $k$ , i.e., the set of sequences which have a support greater than  $\theta_k$ . The set of unique sequences of length  $k$  of a given conversation  $c$  is defined as  $c^k = \{q \in c, |q| = k\}$ . Then, we revisit the problem of protocol discovery as the following:

**Problem 4.1.** *The protocol discovery problem is to find the minimal discovered protocol (DP) (minimal deterministic*

*finite state machine) that 1) accepts all correct conversations  $CC_k = \{c \in CL | \forall q \in c^k, q \in Q_\theta^k\}$ .  $CC_k$  specifies the set of all conversations for which all subsequences of length  $k$  of them ( $c^k$ ) are found in  $Q_\theta^k$  and 2) rejects all conversations  $c \notin CC_k$ .*

This problem is a variation of the well-known problem of regular grammar inference from sample input [6], and it has been proven that there is no efficient algorithm for finding the minimal DP [5]. However, there are two main classes of approaches for finding an approximate for the minimal model in grammar inference. The approaches in the first class start with a specialized model that is precise but usually too complex (e.g., a prefix tree). This model is then iteratively generalized (or simplified) until the desired abstraction level is reached [6], [16]. The approaches in the second class navigate the search space in the opposite direction, starting from a generalized model, which is not precise but simple (e.g., flower model) and which is then specialized to enhance its precision [6]. It has been shown that software interaction models are much closer to generalized models than to specialized ones [17]. Hence, we adopt a specialization-based approach for protocol discovery.

### 4.1 Algorithm Description

Algorithm 1 shows the main steps of the proposed protocol discovery algorithm. The inputs of the algorithm are parameter  $k$  and the set of correct sequences  $Q_\theta^k$ . Its output is a deterministic FSM, representing the discovered protocol (DP). The algorithm is presented in two parts, i.e., the discovery of the protocol model, which consists of four steps shown in Algorithm 1 (shown below), and a complementary part that explains how additional logic are added to this algorithm to handle the log incompleteness (presented in Section 4.2).

**Algorithm 1.** The Protocol Discovery Algorithm.

**Require:**  $k$  (the sequence length),  $Q_\theta^k$  (the set of correct sequences)

**Ensure:**  $DP = (S, s_0, F, M, T)$

- 1: Construct an initial message graph  $G_{initial}$  from  $Q_\theta^k$
- 2: Enhance precision of  $G_{initial}$  based on  $Q_\theta^i$ ,  $3 \leq i \leq k$
- 3: Convert  $G_{enhanced}$  to DP
- 4: Minimize DP

#### 4.1.1 Construction of a Message Graph

In this step, an initial message graph ( $G_{initial}$ ) is constructed from the set of correct sequences  $Q_\theta^k$ , which is obtained from the conversation log  $CL$  after applying noise filtering approach (Section 3). The  $G_{initial}$  has one node for each unique message in the log. For instance, five nodes are created in  $G_{initial}$  corresponding to Cat, PO, Inv, Pay, and Ship for the Retailer example. The directed edges between nodes are established based on sequences in  $Q_\theta^k$ , i.e., there is an edge between two nodes if their corresponding messages follow each other immediately in a sequence  $q \in Q_\theta^k$ . For instance, if  $k = 4$ , for sequence  $q_1 = \langle \text{Cat}, \text{PO}, \text{Inv}, \text{Pay} \rangle$  a directed edge from node Cat to node PO, from PO to Inv, etc. are created. Fig. 5a shows  $G_{initial}$  built from the log of the Retailer service shown in Fig. 3 using  $Q_\theta^4$ . Note that there

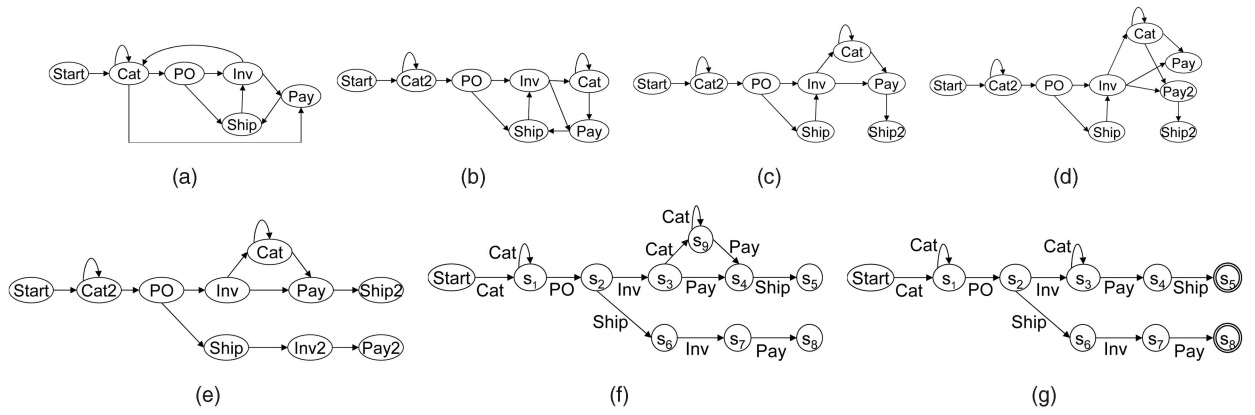


Fig. 5. Applying the discovery algorithm steps on *Retailer* service log. (a) The initial graph  $G_{initial}$  for *Retailer* data set. (b)  $G_{initial,1}$  after splitting node *Cat* in  $G_{initial}$  for  $k = 3$ . (c)  $G_{initial,2}$  after splitting *Ship* node for  $k = 3$ . (d) Splitting only the node next to the last (*Pay*) in  $G_{enhanced,2}$ . (e)  $G_{initial,3}$  after splitting *Pay* node for  $k = 4$  (called  $G_{enhanced}$ ). (f)  $G_{enhanced}$  converted to a FSM. (g) Discovered FSM after minimization.

is no transition from node *Start* to *PO* as the support of sequences with such transition is below  $\theta_4$ .

#### 4.1.2 Graph Precision Enhancement

The graph  $G_{initial}$  is often *overgeneralized*: it accepts, by construction, all sequences of  $Q_\theta^k$  and also some sequences that are not in  $Q_\theta^k$ . For instance, giving an example with sequences of length  $k = 3$ ,  $G_{initial}$  accepts sequence  $\langle \text{Start}, \text{Cat}, \text{Pay} \rangle$ , which is not in  $Q_\theta^k$  (it is an incorrect sequence). This is due to the representation of each message by a single node in  $G_{initial}$ . Hence, when a message appears in two or more sequences with different prefix and suffix messages, its corresponding node accepts all combinations of prefixes and suffixes, some of which may not be correct sequences. For instance, in the above case, the correct sequences are  $\langle \text{Start}, \text{Cat}, \text{Po} \rangle$  and  $\langle \text{Inv}, \text{Cat}, \text{Pay} \rangle$ , in which *Cat* is common. However, it also allows the above incorrect sequence, which is formed by taking the prefix subsequence of *Cat* from the former and its suffix subsequence from the latter correct sequence.

We say that this graph to have a low *precision*, defined as the ratio of correct sequences of a length  $i \leq k$  it accepts over the total number of accepted sequences (modulo loops). The graph has to be modified so that it does not accept incorrect sequences. To disallow such incorrect sequences, separate paths for correct sequences should be created to prevent acceptance of incorrect ones.

We propose Algorithm 2 for enhancing the precision of  $G_{initial}$ . In this algorithm, for each incorrect sequence of length  $k$ , a new path in the graph is created to only allow correct sequences to be accepted. This is also called *splitting* the sequence. To perform this, we start from sequences of the smallest length, i.e.,  $i = 3$  to  $k$  progressively. In this algorithm,  $IS$  stands for the list of Incorrect Sequences. This list is computed in lines 1 to 4.  $il(v, i)$  denotes the list of incorrect sequences of length  $i$  that can be generated from node  $v$  in graph  $G$ . The variable  $prefix$  keeps the set of all unique prefix sequences of length  $i - 1$  in  $IS$ . The purpose of this list is to perform splitting for all incorrect sequences that share the same prefix at once, so to minimize the total number of sequence splittings performed. For instance, for the sequence  $abcd$ , the  $prefix$  is  $abc$ . In fact, a new path for

all incorrect sequences with the same prefix (e.g.,  $abcd$  and  $abce$ ) is created to handle all at once. Node  $seq[i]$  refers to the  $i$ th node in  $seq$ .  $IE$  stands for Incorrect Edges. It determines all nodes  $v$  for which there is an edge from node  $seq[i]$  to  $v$ .

**Algorithm 2.** The Precision Enhancement Algorithm.

**Require:** Initial Graph  $G_{initial}$

**Ensure:** Enhanced Graph  $G_{enhanced}$

```

1: for  $i = 3 \dots k$  do
2:   for each node  $v$  of  $G_{initial}$  do
3:      $IS \leftarrow IS \cup il(v, i)$ 
4:   end for
5:    $prefix \leftarrow$  all unique  $seq[1 \dots (i - 1)]$  in  $IS$ 
6:   for each  $seq \in prefix$  do
7:      $IE \leftarrow$  all  $v', (prefix, v') \in IS$ 
8:     for each node  $seq[j]$  in  $seq[2 \dots (i - 1)]$ ,  $2 \leq j \leq (i - 1)$  do
9:        $seq[j]' \leftarrow$  copy of  $seq[j]$  in  $G$ 
10:    end for
11:    for each node  $seq[j]$  in  $seq[2 \dots (i - 1)]$ ,  $2 \leq j \leq (i - 1)$  do
12:      if  $j > 2$  then
13:        create an edge from  $seq[j - 1]'$  to  $seq[j]'$  in  $G$ 
14:      end if
15:      Copy outgoing edges of  $seq[j]$  to  $seq[j]'$  except the edge from  $seq[j]$  to  $seq[j + 1]$ 
16:    end for
17:    copy outgoing edges of node  $seq[i - 1]$  to node  $seq[i - 1]'$  except for edges to nodes in  $IE$ 
18:    remove edge  $(seq[1], seq[2])$ 
19:    add edge  $(seq[1], (seq[2])')$ 
20:    update  $IS$ 
21:  end for
22:  remove all nodes not reachable from Start
23: end for

```

The creation of the new path (splitting the incorrect sequence) is done by copying all the middle nodes of the incorrect sequence, i.e., nodes  $2 \dots k - 1$  (e.g., second and third nodes in a sequence of length 4), denoted by  $2' \dots (k - 1)'$ . Then, all the outgoing edges of each node  $j$  ( $2 \leq j \leq k$ ) is copied to node  $j'$ , except the edge from  $j$  to

$j + 1$ . Instead, an edge is created from  $j'$  to  $j' + 1$ . The edge between nodes 1 and 2 is removed, and an edge between node 1 and node 2' is created. Note that no edge is created from  $(k - 1)'$  to node  $k$ . This is to disallow the acceptance of the incorrect sequence(s).

To demonstrate how Algorithm 2 works, we apply it on  $G_{initial}$ . First, let us consider the sequence  $\langle \text{Start}, \text{Cat}, \text{Pay} \rangle$ . A copy of the only middle node, i.e., node **Cat** is created and called **Cat2** that has the same outgoing edges except for the one to **Pay** (e.g., to **PO**). Then, the edge from **Start** to **Cat** is removed and an edge from **Start** to **Cat2** is created. The resulting graph is called  $G_{initial,1}$ . It does not accept the incorrect sequence  $\langle \text{Start}, \text{Cat}, \text{Pay} \rangle$  (Fig. 5b). The next incorrect sequence of length 3 is  $\langle \text{Pay}, \text{Ship}, \text{Inv} \rangle$ , which is handled in a similar way by copying **Ship** node. The result graph, called  $G_{initial,2}$ , is depicted in Fig. 5c.

Then, we check for incorrect sequences of the next higher length  $i \leq k$  (here,  $i = k = 4$ ). The sequence  $\langle \text{Ship}, \text{Inv}, \text{Pay}, \text{Ship2} \rangle$  is incorrect. First, a copy of middle nodes **Inv** (called **Inv2**) and **Pay** (called **Pay2**) and also an edge from **Inv2** to **Pay2** are created. Node **Inv2** can get all the outgoing of **Inv**, except edge that goes to **Pay**. In this case, since this edge is the only outgoing edge of **Inv**, no outgoing edges are copied from **Inv** to **Inv2**. The same procedure is followed for node **Pay2**. Next, we remove the edge from **Ship** to **Inv** and create an edge from **Ship** to **Inv2**. Note that in this procedure, we do not create an edge from **Pay2** to **Ship2**. This means that from node **Ship** it is no longer possible to accept the incorrect sequence (see  $G_{initial,3}$  (also called  $G_{enhanced}$ ) in Fig. 5e).

It should be noted that a splitting method for enhancing graphs is also proposed by Cook et al. [11], [18], in the context of workflow model discovery from workflow logs, which recommends splitting the node next to the last node in an incorrect sequence. However, this could not always be applied to the deterministic FSM of service protocols, as it does not allow for removal of the incorrect sequence when more than one node before the last are shared between correct and incorrect sequences. For example, assume the graph of  $G_{initial,2}$  and the incorrect sequence  $\langle \text{Ship1}, \text{Inv}, \text{Pay}, \text{Ship2} \rangle$ . Splitting only the node before the last, i.e., **Pay**, results in the graph shown in Fig. 5d. In this graph, the incorrect sequence still can be accepted. In addition, the node **Inv** is connected to two nodes for **Pay** message, which leads to nondeterminism after the transformation of  $G_{enhanced}$  to FSM.

#### 4.1.3 Conversion into an FSM

In a finite state machine, edges are labeled by message names. The graph produced so far considers messages as nodes. In this step, the graph is converted to an FSM. We convert the enhanced directed graph  $G_{enhanced}$  (Fig. 5e) to an FSM by naming transitions to the name of their target nodes. The resulting graph is a deterministic FSM (Fig. 5f).

#### 4.1.4 FSM Minimization

The resulting FSM may contain equivalent states, i.e., states with the same outgoing transitions to the same target states. This is mainly due to two reasons: 1) the graph constructed in step 1 may not be the minimal form of  $DP$  and 2) splitting may cause the generation of equivalent states, i.e., they have

the same outgoing transitions, which means that they can be merged without changing the FSM properties. Indeed, merging equivalent states allows minimizing the FSM without changing the set of conversations it accepts. For example, in Fig. 5f, states  $s_3$  and  $s_9$  are equivalent and can be merged. We use the Ullman-Hopcroft minimization algorithm [19] for this purpose. Fig. 5g shows the minimized discovered protocol for the *Retailer* service. The final states in FSM are identified by using all the conversations in the log to specify the states, in which most of conversations terminate. The result of this step is the discovered protocol  $DP$ .

## 4.2 Handling Log Incompleteness

Despite the minimization, experiments (see Section 6) reveal that  $DP$ s may be very large, as in absence of some conversations, the algorithm cannot generalize well. The analysis of protocol models shows that there are operations that are *transparent* with respect to a state  $s$ , i.e., their invocation, when in state  $s$ , does not cause a transition out of  $s$ . These are very common in protocols and can be modeled as self-transitions, without requiring additional states. However, since the log is not complete, it may not contain all the distinct message sequences that allow us to infer that the occurrence or nonoccurrence of the operation does not change what can be invoked next. Hence, their modeling requires additional states.

In addition, it often happens that transparent operations are *pervasive*, that is, they can happen in any state, i.e., are transparent in any state. Think, for example, of a search or browse operation while purchasing goods. You can always browse and search, the constraint is that you order before you pay. Transparency helps us to minimize states, while pervasiveness helps us to minimize transitions in the sense that if an operation can occur always and does not affect the protocol, then we can factor it out as opposed to drawing it in the detailed protocol model. Although not as crucial as transparency, this is not a minor benefit, as in real protocol models failing to recognize pervasive operations cause the model to be cluttered with arcs and, hence, is hard to read.

To handle this issue, we propose the following heuristics: We look for transparent and pervasive operations in the initial directed graph  $G_{initial}$ , in which each node corresponds to an operation, in two steps. In the first step, we identify nodes in  $G_{initial}$  with incoming edges from more than half of the service operations. Such operations are considered as *candidates* for pervasive operations. In a second step, we apply the following condition for each candidate operation  $o$ , identified in the first step: If  $Pred(o)$  represents the set of predecessors of  $o$  in  $G_{initial}$  (i.e.,  $\forall p \in Pred(o)$ , there is an edge from  $p$  to  $o$  in  $G_{initial}$ ), and  $Succ(o)$  represents the set of successors of a node  $o$  (i.e.,  $\forall t \in Succ(o)$ , there is an edge from  $o$  to  $t$  in  $G_{initial}$ ), then we should have  $Succ(o) = Succ(p)$ . This condition implies that appearance of operation  $o$  after  $p$  does not change its successor operations. We say that  $o$  is a transparent operation with respect to  $p$ .

Given an incomplete log, meeting the above condition is rarely possible for operations  $o$  and  $p$ . Therefore, we use a looser version of this condition that requires  $\forall t \in Succ(o)$ ,  $t \in Succ(p)$ , but the reverse condition  $\forall t \in Succ(p)$ ,  $v \in Succ(o)$  should hold for at least 90 percent of operations in

$Succ(p)$ . This is an approximation to handle incompleteness of the log and to allow  $p$  to have few successor operations that are not successors of  $o$  in  $G_{initial}$ . After discovering all transparent operations  $o$  and their corresponding predecessors  $p$ , we remove the edges from  $p \in Pred(o)$  to  $o$  in  $G_{initial}$ . Then, we apply steps 2 and 3 of the algorithm (splitting and converting to FSM) on  $G_{initial}$ . Finally, we put back edges that we removed, as self-transitions labeled with operation  $o$  from node  $p$  to  $p$ ,  $\forall p \in Pred(o)$ . This means that operation  $o$  is transparent for each  $p$ . If an  $o$  is transparent for all operations  $p \in Pred(o)$ , then  $o$  is called a pervasive operation in  $DP$ . In a last step, we convert  $DP$  to an FSM and minimize it. Application of these heuristics considerably improves the size of  $DP$  and allows for compensating the imperfection related to incomplete logs.

### 4.3 Time Complexity of the Algorithm

The initial graph is built in step 1 with one node for each of the  $p$  different operations involved in the protocol. Then, this graph is connected according to the sequences of length  $k$  that are found in the set of correct sequences  $Q_{\theta}^k$ . This is done in a single pass through the set  $Q_{\theta}^k$ . Step 1 therefore has a complexity in  $O(|Q_{\theta}^k|)$ .

In order to estimate the complexity of step 2, we need to see how many sequences are generated from the initial graph. We used in Section 3 the notation  $\phi$  to represent the average number of operations that can follow a given operation in the log. The initial graph  $G_{initial}$  is not built from all those sequences but only from the subset of correct sequences (those in  $Q_{\theta}^k$ ). Thus, the average arity of each node in  $G_{initial}$  is less than  $\phi$ . We denote by  $\varphi$  ( $\varphi \leq \phi$ ) the average arity of nodes in  $G_{initial}$ . Using this notation, the number of sequences of length  $k$  generated in step 2 is given by  $p\varphi^{k-1}$ .

Step 2 performs splitting operations (nodes) by generating sequences of length  $2 < i \leq k$  and checking if they are present or not in the list of correct sequences of length  $i$  ( $Q_{\theta}^i$ ). A total of  $p\varphi^{i-1}$  sequences are thus generated and looked up in the list  $Q_{\theta}^i$ . The lookup operation performs very fast since sequences in  $Q_{\theta}^i$  are stored in an indexed structure with  $p$  entries, corresponding to the  $p$  operations that form those sequences, and with  $i$  levels. Each lookup then is performed in  $O(i)$ . Since we have,  $i \leq k$ , the complexity of this operation can be bounded by  $O(k.p.\varphi^{k-1})$ .

Step 3 does not modify the graph that results from step 2. Finally, in step 4, a minimization of the enhanced graph is performed. The time complexity of the minimization algorithm we use is  $O(s^2)$ , in which  $s$  is the number of nodes [19]. In order to compute the complexity of step 4 in terms of our data size (number of conversations), we need to estimate the number of nodes in  $G_{enhanced}$ . We recall that it is built from  $G_{initial}$  by performing split operations. Each split operation adds at most  $k - 2$  nodes to the  $p$  present in the initial graph. (To be precise, each split operation creates  $i - 2$  nodes, where  $i$  varies from 2 to  $k$ .) We can estimate the size of the enhanced graph—the graph obtained after all split operations are applied—by  $|G_{enhanced}| = p + (k - 1).\delta$ , where  $\delta$  represents the number of split operations. The number of split operations is determined by the number of sequences generated by  $G_{initial}$  that do not belong to  $Q_{\theta}^k$ , a rough estimate of which is given by  $\delta = p\varphi^{k-1} - |Q_{\theta}^k|$ .

Actually, from what precedes, we have  $|Q_{\theta}^k| \approx p\varphi^{k-1}$ , which shows that  $\delta$  is necessarily small. This is natural since the initial graph was connected using precisely the set of correct sequences  $Q_{\theta}^k$ . Estimating more precisely the number of split operations would require introducing additional criteria on the data set, which would not be very intuitive. The time complexity of step 4 hence is  $O((p + (k - 2).\delta)^2)$ .

Summing all those steps gives the complexity of the discovery algorithm. It shows that the algorithm polynomially depends on the maximum sequence length  $k$ , the number  $p$  of operations involved in the protocol, and the average number  $\phi$  of operations that follow a given operation. However, we can see that the size of the data set does not impact the time complexity of the algorithm but only that of the noise identification algorithm presented in Section 3. Experiments in Section 6 show that the algorithm is practically scalable with regard to  $k$ ,  $p$ , and  $\phi$  in real data sets.

## 5 USER-ASSISTED PROTOCOL REFINEMENT

It is unlikely for automated approaches to discover the correct protocol model from an imperfect service log. Due to the presence of noise and the fact that a frequency threshold often does not provide a clear-cut separation between noisy and correct conversations, the discovered protocol  $DP$  may erroneously accept conversations that should not be accepted, or vice versa. It may not accept some conversations that it should, because they are infrequent and hence identified as noisy. Therefore, users may require refining the protocol discovered after the previous phase.

The protocol refinement is challenging as in large logs the number of noisy conversations may be high, even though noise is random and rare. Therefore, it is not realistic to ask users to manually go through all of them and check if they should be accepted or not by the protocol. Hence, the problem is how to guide users through the interactive analysis of conversations, which are identified as noisy. We call such conversations *uncertain* as we are not sure if they are really noisy. To overcome the above issues in discovered protocols, we provide interactive protocol refinement mechanisms by featuring 1) *metadata driven protocol refinement* and 2) *distance-based interactive protocol refinement*.

### 5.1 Metadata Driven Protocol Refinement

The goal of this step is to improve the precision of discovered protocols by disallowing all conversations that are accepted by mistake. We annotate the discovered protocol model with various metadata including transition support (i.e., the percentage of conversations in the log that traverse a transition), final state support (i.e., the percentage of conversations that terminate in a state relative to the number of conversations that traverse it), and protocol support (i.e., the percentage of conversations in the log that are accepted by the protocol). The graphical interface of the tool (Fig. 6) enables users to visually browse the discovered protocol and examine associated metadata to understand potential errors made during the discovery process and

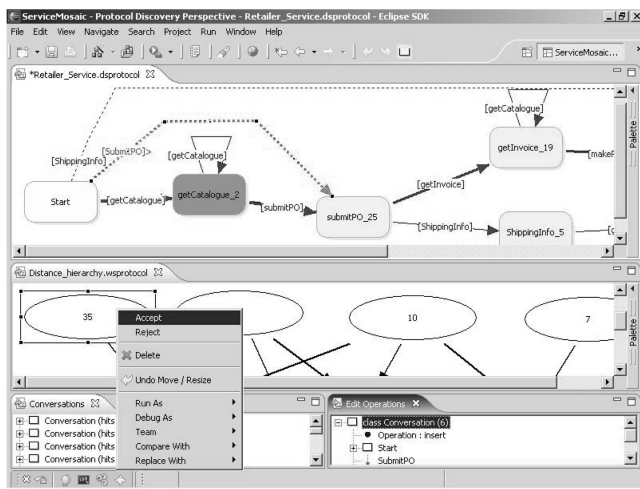


Fig. 6. Discovery and refinement editor (the names of states are generated as a combination of the name of an incoming message to that state and a number to guarantee the uniqueness of state names).

correct them (the metadata window is not shown due to space limitations).

The tool facilitates the protocol browsing by identifying regions of *DP* that requires more attention in the sense that we are less confident about their correctness (showed by dashed lines in Fig. 6). This is performed by highlighting different levels of transition supports (final state supports, respectively) using various arrow thickness so that weakly supported transitions are represented with thinner lines (and brighter colors, respectively). The stronger a line, the more we are certain about it in the model; the darker a state, the more we are confident that it is a final state. The users can inspect states, based on their color, and edit the final state property of the states to (un)mark them as final states. This allows us to address the issue of “partial conversations” during the refinement step.

## 5.2 Distance-Based Interactive Protocol Refinement

The goal of this step is to help users in allowing correct (but infrequent) conversations in *DP* that are excluded from it. To tackle this problem, we analyze uncertain conversations in the log based on measuring the *distance* between discovered protocol *DP* and an uncertain conversation. This is because conversations are considered as strings of operation names, and *DP* is as an accepter for all correct conversations. The possible types of differences between uncertain conversations and *DP* are missing messages, messages swapping, or additional messages. These correspond to various types of infrequent sequences that may be correct. However, the first two types are also created due to noise (Section 3). This is one of the reasons that the refinement step should be user driven to distinguish between these cases.

We adopt the approach for computing *edit distance* between strings [20], in which differences between strings are of type of missing, additional and swapped characters, to compute distance between a conversation and *DP*. In a nutshell, this distance is computed by counting the number of *manipulation operations* that have to be performed on the conversation string to obtain a string that is accepted by

*DP*. For example, a conversation may be accepted by *DP* if we manipulate it by adding message  $m_x$  between  $m_1$  and  $m_2$ . In this case, we applied one operation so the distance is 1. The reason why the original conversation is not accepted can be either because the sequence  $m_1, m_2$  is infrequent (but correct) or because the correct sequence is instead  $m_1, m_x$ , and  $m_2$ , but  $m_x$  was not captured due to noise. In the latter case, no change is needed to *DP*, but in the first one, *DP* should be amended to allow for the generation of the  $m_1, m_2$  sequence.

We also observe that, typically, the same type of correction (e.g., insertion of  $m_x$ ) may be proposed by several infrequent conversations that are accepted by *DP*. Indeed, the number of different corrections that can be needed is often small compared to the number of conversations that would require corrections to be accepted. Furthermore, the higher the number of conversations in which a given manipulation operation is needed, the more likely that the corresponding (original, none manipulated) sequence is relatively infrequent but in fact correct. The combination of the above observations leads to the idea of managing the refinement process by 1) guiding users through analyzing possible manipulations (small in number) instead of the possible conversations (very high in number) and 2) ranking manipulations based on how often they occur.

In order to perform the above strategy, we need to 1) identify the possible operators that can transform conversations to be accepted by *DP* and 2) identify the best paradigm to guide users in walking through the different manipulation operations so to quickly identify which conversations are infrequent but correct and which are instead noisy. These aspects are discussed below.

### 5.2.1 Edit Distance and Manipulation Operators

The notion of edit distance has been studied in the past, mainly by Levenshtein, and here, we apply the Levenshtein edit distance [20], which is based on three operations: insertion, substitution, and deletion. Specifically, for conversations, we consider operations  $swap(c, m_1, m_2, s)$ , which inverts the order of two messages  $m_1$  and  $m_2$  in a conversation string  $c$ , where  $m_1$  occurs with the conversation in state  $s$ ;  $insert(c, m_1, m_2, m_x, s)$ , which inserts message  $m_x$  between  $m_1$  and  $m_2$  in conversation  $c$ , again taking the occurrence of  $m_1$  in state  $s$ ; and  $delete(c, m_x, s)$ , which removes message  $m_x$  in conversation  $c$  whenever it is in state  $s$ .

The typical approach for computing Levenshtein distance is through dynamic programming. This leads to a quadratic time complexity on the length of the conversation. In our context, we compute an approximate distance by leveraging heuristics that yields a linear time. In particular, swap is tried first, insertion next, and the delete the last. This order corresponds to the biggest number of corrections that are accepted. In fact, swap treats the case where a pair of operations  $a$  and  $b$  could be invoked in any order with respect to each other, i.e., both  $ab$  and  $ba$  are possible, however, one of the orders is more common in the log. Insert operation handles the cases where it is believed that the logging infrastructure missed logging a message  $b$  in a given sequence  $ac$  (so it should have been  $abc$ ), while  $ac$  is

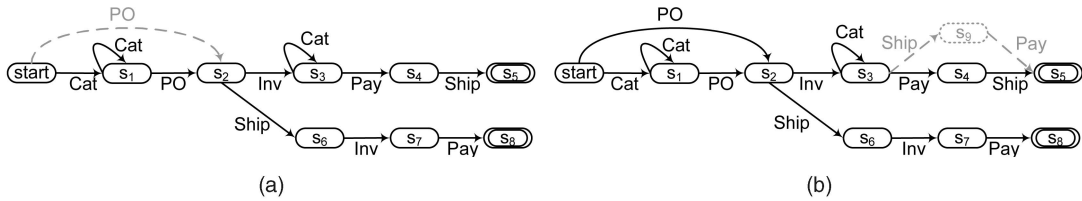


Fig. 7. Refinement operations on *DP*. (a) Insert operation (Cat) in the sequence (Po, Inv, Pay, Ship). (b) Swap operation (Ship and Pay).

also correct. Conversely, the delete operation compensates for cases where it is possible to have conversation *abc*, while in the model, only *ab* is allowed. However, the simplified algorithm does not always guarantee returning the minimum number of manipulation operations for a given conversation. This is an acceptable approximation in our case as the intuitions behind the heuristics have been confirmed by experiments.

**Example 5.1.** Consider the protocol *DP* discovered from the log of the Retailer service in Fig. 5g and conversation  $\langle \text{Po, Inv, Pay, Ship} \rangle$ . The distance of this conversation with *DP* is 1, since if we insert a *Cat* operation in the beginning of the conversation, it becomes accepted by *DP*.

### 5.2.2 Refinement Operations Corresponding to Manipulation Operations

To enable refining discovered protocol models based on manipulation operations on conversations, we need to define a set of corresponding refinement operations on the protocol. For example, corresponding to an insert operation in a conversation (e.g., for conversation  $\langle \text{Po, Inv, Pay, Ship} \rangle$ ), we should have an add transition operation that allows adding a transition from the state *Start* to the state *s<sub>2</sub>*, as in Fig. 5g (see Fig. 7a). If the user accepts this change, the transition becomes part of the model (see Fig. 6). Refining based on a delete operation translates in adding a transition, too. For example, for conversation  $\langle \text{Cat, Po, Inv, Pay, Cat, Ship} \rangle$ , deleting *Cat* make it accepted by *DP*. If such a conversation is considered correct, then refining *DP* translates in adding a self-transition in state *s<sub>4</sub>*. For a swap operation, we need to add a state and two transitions. For instance, conversation  $\langle \text{Cat, Po, Inv, Ship, Pay} \rangle$  suggests a swap between *Ship* and *Pay* from state *s<sub>3</sub>* in Fig. 5g. To refine the protocol based on such swap operation, we need to create a new state, e.g., *s<sub>9</sub>* and two transitions, one from state *s<sub>3</sub>* to *s<sub>9</sub>* for *Ship* operation, and the other from state *s<sub>9</sub>* to state *s<sub>5</sub>* (see Fig. 7b). These considerations make it clear that we need two refinement primitives on the protocol, one for adding states and the other for adding transitions.

### 5.2.3 Classification of Uncertain Conversations

Once we have computed the distance for each conversation, the manipulation operations on each and also the refinement primitives corresponding to each manipulation operations, we construct an *edit distance hierarchy* whose nodes represent the application of one or more edit operators, regardless of the conversation (Fig. 8). For example, a node can be *swap*(., *m<sub>1</sub>*, *m<sub>2</sub>*, *s*).

The cardinality of the node is represented by the number of conversations to which, if we apply the operators

associated to the node, the conversation is transformed into one accepted by the protocol. Leaf nodes of the hierarchy are associated to only one edit operation and, therefore, identify conversations of distance 1 from the protocol. At higher levels, nodes are associated to progressively more operations. A parent node includes all the operations of its children nodes. The rationale behind this hierarchy is that we can guide the user from the bottom to the top of the hierarchy. The cardinality of each node gives user an intuition of what to consider first: The operations that are required more often are more likely to affect conversations that have been incorrectly classified as noisy and hence to be allowed by the protocol model.

Finally, we provide a visual interactive approach that by selecting each node in the hierarchy highlights the refinement operations on *DP* for the user (Fig. 6 shows the case in Fig. 7a). The user can browse this hierarchy starting from the leaves, visually examine the proposed refinement on the protocol, and state whether the refinement operations on the protocol are correct (the sequence was not noisy) or if instead the sequence was noisy. For example, refinement operations based on  $\langle \text{Po, Inv, Pay, Ship} \rangle$  allow inclusion of such conversation in *DP*, and it is associated to the highest cardinality node (bottom part of Fig. 6). On the other hand,  $\langle \text{Cat, Po, Inv, Ship, Pay} \rangle$  may not be accepted as it is not supported by the designed protocol of Retailer, since it is the result of a swapping order error. Typically, the large majority of the conversations incorrectly classified as noisy are accounted for by exploring high cardinality nodes.

## 6 EXPERIMENTS AND VALIDATION

### 6.1 Quality of Discovered Protocols

To measure the performance of the algorithm, in experimental settings, we assume that the reference model *P* (the model actually supported by the service) is known. We use the classical *recall* and *precision* metrics [21]. Recall is the

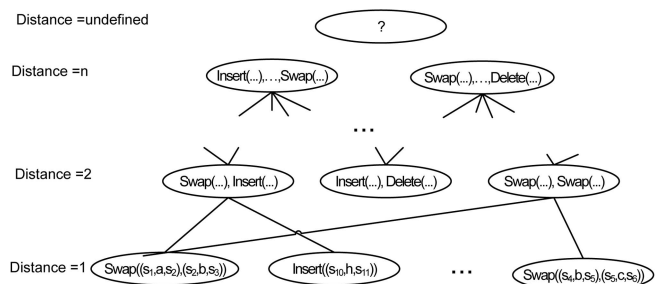


Fig. 8. The class hierarchy for the classification of uncertain conversations.

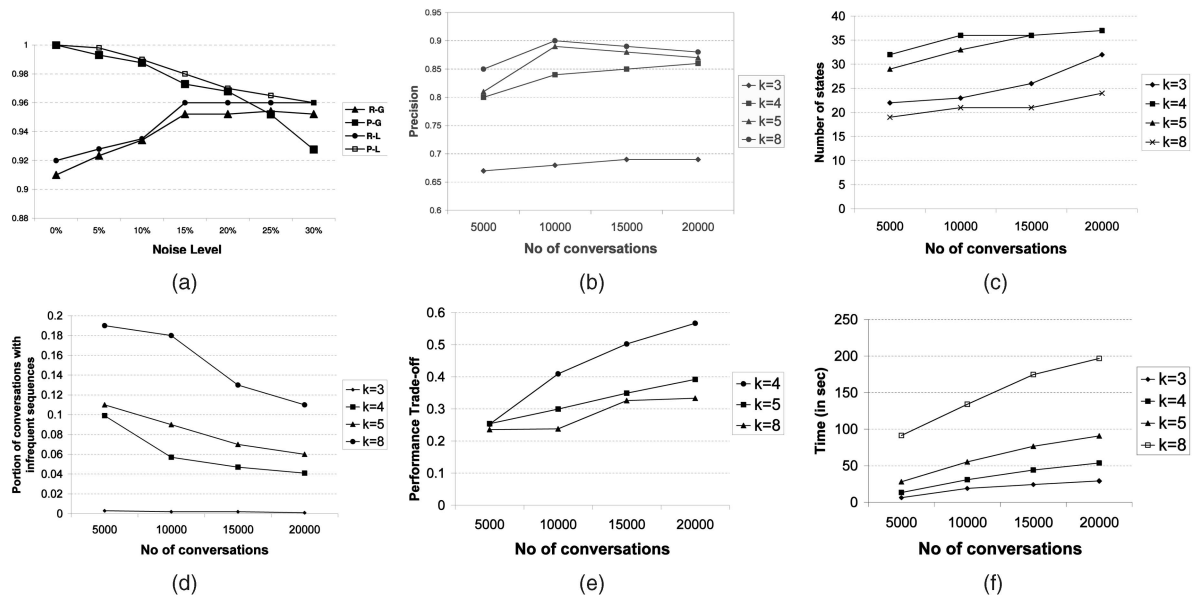


Fig. 9. Results of experiments: (a) on the *Retailer* data set and (b), (c), (d), (e) and (f) on the *Robostrike* data set. (a) Precision and Recall of *DP*. (b) Acceptance rate of *DP*. (c) Number of states of *DP*. (d) Conversations in the testing set, having sequences with supports less than estimated noise threshold. (e)  $k = 4$  achieves the best performance trade-off. (f) Execution time of the algorithm.

percentage of the correct conversations (i.e., conversations accepted by the reference protocol  $P$ ) that are also accepted by  $DP$ . Precision is the number of correct conversations accepted by  $DP$  divided by the number of all conversations in the log accepted by  $DP$ . We also consider the size of discovered protocols as a measure of its quality. Protocols with higher precision and recall and smaller size are desired.

However, in real-world settings,  $P$  is unknown, and we need to revisit definitions of precision and recall. In this context, we use the classical machine learning approach of  $k$ -fold cross validation [21] to split our data sets into learning and testing sets. Then, we define the precision to be the percentage of conversations in the testing set that are accepted by  $DP$ . We do not use recall measure, as it is not clear how to measure it in this setting.

## 6.2 Data Sets

We perform experiments on synthetic and real-world data sets:

*Synthetic data set.* We simulated the *Retailer* service (see Fig. 3 for a simplified representation of the *Retailer* service protocol) to collect the log of its interactions with clients. For this data set, although the scenario is synthetic, the log is collected using a real-world commercial logging system for Web services (HP SOA Manager). Table 1 shows the characteristics of the data set. We have implemented *Retailer* service in Java utilizing Apache Axis as the SOAP implementation and Apache Tomcat as the Web application server. Ten Java service clients are deployed to concurrently interact with the *Retailer* service. This data set (consists of 5,000 conversations) was imperfect by containing swapped and missing messages and also incomplete conversations caused by clients that left conversations or crashed. We analyzed the log and identified conversations that are not accepted by  $P$  and computed the noise level as reported in Table 1.

*Real-world data set.* The second data set is a real-world log of interactions of a multiplayer online game Web service called *Robostrike* ([www.robostrike.com](http://www.robostrike.com)) with its clients. The service simultaneously manages ongoing conversations with players as clients. The service has 32 operations, which clients invoke by sending synchronous or asynchronous XML messages. In this data set, conversations range from very short (1 message) to very long (1,921 messages), depending on the time that clients spend on playing. The conversations captured in the log can be noisy (as discussed before) and incomplete (e.g., a client may disconnect at any time). Table 1 presents the statistics of this data set. We collected 25,804 conversations over a period of two weeks. This data set allows us to evaluate the capabilities of our approach in discovering precisely unknown and complex protocols and its scalability. To apply  $k$ -fold cross validation, we split conversations of *Robostrike* into five sets, and in each experiment, we used four sets for the learning and the last one for the testing.

## 6.3 Robustness of Noise Identification Approach

The precision and recall of the algorithm on the *Retailer* data set is presented in Fig. 9a for zero to 30 percent of noise level (shown with P-G and R-G, respectively). In fact, the actual log contains 9.78 percent of noise, and we artificially introduced more random noise (of type of swapping or removing one or more messages in a conversation) to go up to 30 percent. As it can be seen, the level of the noise in the log does not sensibly affect the precision of  $DP$  as it always stays above 90 percent. However, the recall never reaches 100 percent as some correct, but infrequent sequences are classified as noisy based on estimated  $\theta$ . This is one of the reasons for having a refinement step. We used  $k = 4$  as the sequence length in these experiments. We have also performed experiments using this data set to compare our approach with the (rule-) learning-based approach for noise identification in process logs [7] (precision and recall are

shown with P-L and R-L, respectively) It shows that our approach achieves almost similar results in data sets with up to 20 percent of noise, which represent most practical cases. However, that approach achieves a better performance for data sets with a higher level of noise (e.g., for 30 percent noise, P-G is 92.5 percent, while P-L is 96 percent). The result of our approach is acceptable for most practical cases. In addition, it is purely statistical (does not need training).

The experimental results also indicate that the approach is highly scalable in the number of conversations, depicted in Fig. 9f. The runtime increases almost linearly by the conversation size.

#### 6.4 Handling Incompleteness in the Service Log

The direct application of the algorithm (steps 1-4) on Robostrike data set results on the average precision of 88.7 percent and the size of 67.8 states ( $k = 4$ ). Although the precision is very good,  $DP$  is conceived to be complex for a protocol designer to work with due to the high number of states. Then, we applied our heuristics on transparent and pervasive operations on this data set. The result shows that out of 32 operations of the service, nine are transparent in many states of  $DP$ , and three are pervasive. The  $DP$  discovered after applying our heuristics has 37 states. This  $DP$  also achieves average precision of 86.3 percent; however, it is simpler, with less number of states.

#### 6.5 Impact of the Sequence Length $k$ on the Performance

To evaluate the performance of the algorithm with different  $k$  values, we executed the algorithm for values  $k = 3, 4, 5$ , and 8 on Robostrike data set with different number of conversations (5,000 to 20,000). In all of these experiments, we also applied our approach for handling incompleteness. Fig. 9b shows the precision values of  $DP$  in these experiments. Fig. 9c illustrates the number of states for  $DP$ s. Fig. 9d depicts the portion of conversations in the testing data set that are estimated to contain at least one noisy sequence of length  $k$ . We define a performance trade-off as an indicator of a good  $k$  value as follows: A  $k$  value is the best if using it results in a model with a high precision, small size (denoted by  $Size_{DP}$ ), and relatively smaller portion of conversations that are filtered based on noise estimation (denoted by  $n_k$ ). Putting all together, we need to maximize  $Precision_{DP}/(Size_{DP} * n_k)$ . Fig. 9e illustrates values of this performance trade-off for different values of  $k$  (we excluded  $k = 3$  from comparison as it achieves a low precision).

This figure suggests that  $k = 4$  best meets this performance trade-off. The result is consistent with the intuition as well. Using  $k = 3$ , we underestimate noise (a small portion of sequences is identified as noisy), and so, we split less sequences. This results in a small model that has a good recall but low precision, as the variability of sequences cannot be captured with this value of  $k$ . For higher  $k$  values ( $k = 5, 8$ ), the precision gets better and better but at the expense of generality of the model. In fact, for such  $k$  values, many of sequences are considered as noisy, so we build the model out of small number of sequences. Therefore, we end up with a small model that does not have a good recall but achieves a high precision. While we recommend value of  $k = 4$ , if any

particular factor, e.g., the size or precision is more important for a discovery task, the user can use above behavior of the algorithm as a guideline to set the appropriate  $k$  value.

### 6.6 Refinement of the Discovered Protocol

#### 6.6.1 Metadata Driven Refinement

The discovered protocol  $DP$  for Retailer includes a transition called ShippingInfo from state Start to state ShippingInfo (see Fig. 6). However, this transition should not be included in  $DP$  as it is not a part of reference model  $P$  in Fig. 3. It is introduced because the sequence (Cat, Ship, Inv, Pay) is mistakenly classified as a correct sequence by the noise estimator, as the support of this sequence, in which Inv and Ship messages are disordered, is 0.025 (see Fig. 4a). This value is above the estimated noise threshold, i.e., 0.018, for sequences of length 4 in this data set. Computing the metadata for this transition in  $DP$ , its support is  $supp = 0.0025$  (because some conversations that contain this sequence also contain other incorrect sequences, so that they are not accepted by the model). This support is below the noise threshold. We identify this transition as “weak” and display it with dashed line in the model.

#### 6.6.2 Distance-Based Protocol Refinement

The result of analyzing uncertain conversations for Retailer data set shows that the edit distance hierarchy has 16 first level classes (leaves) and a total of 25 classes. Classes at each level are ranked based on the number of conversations that will be accepted by  $DP$  by applying the proposed changes. Selecting the highest cardinality class in the leaves suggests adding the transition SubmitPO from state Start to state SubmitPO in the protocol (see Fig. 6). This is a desired suggestion (see Fig. 3), however, it has been excluded from  $DP$  during discovery due to low support of relevant sequences. By inspecting the first five classes with the highest cardinalities in the hierarchy, 83 percent (5 out of 6) of the transitions that were missing in  $DP$  of Retailer were examined for inclusion. This number is very small compared to 421 conversations of Retailer that would have been examined without using the distance hierarchy. For the Robostrike data set ( $k = 4$ ), there were 1,947 uncertain conversations. The class hierarchy for these conversations consists of 55 classes in seven levels. The first level (leaves) has 29 classes. The five highest cardinality classes contain 262, 237, 152, 130, and 126 conversations. Our refinement approach reduces the task of inspecting 1,947 conversations manually to inspecting high cardinality classes at seven levels, which is small relative to the number of uncertain conversations.

## 7 RELATED WORK

We study related work with respect to the three stages of protocol discovery: noise identification and handling, protocol discovery, and refinement.

### 7.1 Noise Identification and Handling

In the area of model discovery, the existence of noise in data is recognized in process discovery research [7], [9], [11], [12], [13]. Most of process discovery approaches use a

frequency threshold to filter noise in a preprocessing step [9], [11], [12], [13]. This is performed by manually setting a frequency threshold. In particular, the approach of Cook et al. [11] assumes manual provision of a cutoff threshold. Agrawal et al. [12] compute the noise threshold based on assumption that error rate of the logging infrastructure is given. Van der Aalst et al. [13] assume a noise factor with the default value of 0.05. In [9], the need for automatically learning it in process logs is acknowledged.

In this paper, we have proposed an approach to automatically estimate the noise threshold from the input data set. This is based on analyzing the frequency distribution of subsequences of conversations. Sequence decomposition has been also explored in other contexts, e.g., in clustering of log traces based on frequent subsequences [22]. Our approach is significant since the 1) user generally has no idea of how to manually set a threshold for a given data set, and 2) the value of the threshold is not the same for all data sets and for all sequence lengths.

Our noise identification algorithm could be seen as an outlier detection approach [23]. However, traditional outlier detection are based on supposing a statistical model that would fit the data, based on observing the deviation of data, or based on minimizing the entropy of the data set. In our problem, if an error occurs, it will likely have an effect on a series of sequences, which will have a similarly low-frequency count. For this reason, instead of directly identifying outliers, we seek to identify a frequency threshold and will declare all sequences that are less frequent than this threshold as outlier. This is consistent with existing work in this area, which specify thresholds manually [11], [12], [13].

The approach presented in Maruster et al. [7] uses a machine learning-based approach to learn a set of robust rules from potentially noisy data set. Each rule specifies how to identify the relationship (e.g., sequential, parallel, exclusion, etc.) of a pair of activities in a workflow log by inferring the values of a set of frequency-based parameters. The learner is trained with a set of labeled activity pairs and corresponding frequency statistics from a noisy log. This approach achieves a high precision, as reported in Section 6.3. However, preparing training samples is hard in real-world scenarios. This requires analysis of data from a similar/same-logging infrastructure to understand which sequences are noisy and what kinds of mistakes the logging infrastructure makes. Nevertheless, our approach is statistical and analyzes the frequency distribution of the input data set to discover a noise threshold.

Finally, probabilistic models [9], [16], [24] provide a natural option for protocol discovery applications and noise handling. However, given that we propose to identify noise before the model discovery step to avoid the disturbance of noise and the need for deterministic behavior of services, a deterministic model such as the one adopted in this paper is well suited.

## 7.2 Model Discovery Techniques

*Sequential process models discovery.* Cook et al. [11], [18] propose a specialization-based approach for discovering sequential software process models represented as non-deterministic FSMs. For specialization, Cook et al. approach

recommends splitting the node next to the last in an incorrect sequence. Our experiments with this approach (illustrated also through an example in Section 4) show that it does not allow us to remove some incorrect sequences. We have presented a new splitting algorithm that recommends splitting all the middle nodes of an incorrect sequence to avoid such cases. Herbst and Karagiannis [17] also present a specialization-based approach for discovering sequential model of process logs represented using Hidden Markov Model (HMM). In this approach, all states with more than one incoming are considered as splitting candidates and so examined for splitting into two states with disjoint set of inputs to see if any of such result models improves the log-likelihood with a value above a user-specified threshold. Therefore, the number of potential examinations for each state is exponential to the number of incoming transitions of that state. The complexity of our splitting method is polynomial in the order of number of sequences of length  $k$  generated from the graph  $G$  (see Section 4). In addition, our approach does not need a user-defined threshold for model discovery, which may be difficult to set manually.

In software engineering, researchers analyze the traces of software executions to discover execution models of software that is mainly used for debugging of the code [10], [25]. The authors in [25] use existing probabilistic automata learners to discover data and control flow specifications of the program, and so, they do not offer a new discovery approach. In [10], they limit the scope of specification discovery to finding frequent patterns (sequences of events) but not a complete model.

*Concurrent process models discovery.* Another related area is discovering concurrent process models from workflow logs [9], [12], [13], [18], [22], [26], [27] (some offered in ProM toolset [28]). These approaches allow for discovering process models with complex constructs such as parallelism and synchronization points. However, they make the assumption that each activity appears only once in the target model [13]. In contrast, it is quite common for the same activity to appear in different part of the model, e.g., operation Ship in Retailer example. Our experiments with those approaches presented in ProM toolset also confirms that discovered models for protocol data sets (e.g., Retailer data set) are overgeneralized since they do not perform splitting.

In addition, to the best of our knowledge, no approaches in model discovery consider handling log incompleteness. Our approach exploits statistical properties of messages in the log to predict some missing service conversations to allow for the generalization of the discovered models and so to compensate for log incompleteness. The work of Dustdar and Gombotz [14] uses existing process discovery techniques [13] to discover the composition logic of a Web service.

## 7.3 Model Refinement

The idea of interactive mining of models is previously explored in process mining applications [29]. However, our protocol refinement approach is original, and to the best of our knowledge, no support is provided for refinement of discovered models to compensate for log imperfection.

Existing approaches are limited to simple editors for model visualization, e.g., an editor for manual manipulation of discovered models [11]. In the area of software engineering, the problem of debugging software specifications is studied in [30]. In this approach, after discovering software specifications from software execution traces [25], a model checking tool is applied on the source code of the software to generate all the execution traces of the software that violate the discovered specification. In the context of Web services, if the source code of service is available, this approach could be complementary to our approach, as it allows to obtain a complete log (by generating all possible conversations of service) and also to obtain additional correct traces that could be used to refine the discovered protocol model. Finally, tools offered for viewing and manipulating process models in the area of management science (e.g., [31]) are complementary to our tool, as the discovered protocol models can be then maintained with respect to other process models in the enterprise.

## 8 CONCLUSION AND FUTURE WORK

The approach presented in this paper addresses the problem of discovering protocol models from real-world service conversation logs, which are often imperfect. We have characterized the possible imperfections of logs in real-world setting. In particular, we have presented a quantitative approach and an algorithm for estimating a noise threshold used to filter noisy conversations from the log. We have proposed a protocol discovery algorithm that is robust to log incompleteness and an interactive protocol refinement technique to support correction of the discovered protocols. We have presented the implementation of the system. Finally, we have validated the proposed techniques using both synthetic and real-world service logs. As part of future work, we are working on the problem of correlation of messages into conversations in service interaction logs.

## REFERENCES

- [1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services—Concepts, Architectures, and Application*. Springer, 2004.
- [2] B. Benatallah, F. Casati, and F. Toumani, "Representing, Analysing and Managing Web Service Protocols," *Data and Knowledge Eng. J.*, vol. 58, no. 3, pp. 327-357, 2006.
- [3] H. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, and F. Casati, "Protocol Discovery from Web Service Interaction Logs," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, 2007.
- [4] H. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, F. Casati, F. Toumani, and J. Ponge, "Servicemosaic: Interactive Analysis and Manipulations of Service Conversations," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, 2007.
- [5] E.M. Gold, "Complexity of Automaton Identification from Given Data," *Information and Control*, vol. 37, no. 3, 1978.
- [6] R. Parekh and V. Honavar, "Grammar Inference, Automata Induction, and Language Acquisition," *A Handbook of Natural Language Processing*, chapter 29, 2000.
- [7] L. Maruster, A.J. Weijters, W.M. Aalst, and A. Bosch, "A Rule-Based Approach for Process Discovery: Dealing with Noise and Imbalance in Process Logs," *Data Mining and Knowledge Discovery*, vol. 13, no. 1, 2006.
- [8] W. Pauw et al., "Web Services Navigator: Visualizing the Execution of Web Services," *IBM System J.*, vol. 44, no. 4, 2005.
- [9] R. Silva, J. Zhang, and J. Shanahan, "Probabilistic Workflow Mining," *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, 2005.
- [10] J. Yang et al., "Perracotta: Mining Temporal API Rules from Imperfect Traces," *Proc. Int'l Conf. Software Eng. (ICSE)*, 2006.
- [11] J.E. Cook and A.L. Wolf, "Discovering Models of Software Processes from Event-Based Data," *ACM Trans. Software Eng. and Methodology*, vol. 7, no. 3, 1998.
- [12] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining Process Models from Workflow Logs," *Proc. Int'l Conf. Extending Database Technology (EDBT)*, 1998.
- [13] W. van der Aalst et al., "Workflow Mining: A Survey of Issues and Approaches," *Data and Knowledge Eng. J.*, vol. 47, no. 2, 2003.
- [14] S. Dustdar and R. Gombotz, "Discovering Web Service Workflows Using Web Services Interaction Mining," *Int'l J. Business Process Integration and Management*, vol. 1, no. 4, 2006.
- [15] D.E. Knuth, *The Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley, 1997.
- [16] F. Thollard, P. Dupont, and C. Higuera, "Probabilistic DFA Inference Using Kullback-Leibler Divergence and Minimality," *Proc. Int'l Conf. Machine Learning (ICML)*, 2000.
- [17] J. Herbst and D. Karagiannis, "Integrating Machine Learning and Workflow Management to Support Acquisition and Adaptation of Workflow Models," *Int'l J. Intelligent Systems Accounting, Finance and Management*, vol. 9, no. 2, 2000.
- [18] J. Cook, Z. Du, C. Liu, and A. Wolf, "Discovering Models of Behavior for Concurrent Workflows," *Computers in Industry*, vol. 53, no. 3, 2004.
- [19] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1990.
- [20] R.A. Wagner and M.J. Fischer, "The String-to-String Correction Problem," *J. ACM*, vol. 21, no. 1, 1974.
- [21] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2004.
- [22] G. Greco, A. Guzzo, and L. Pontieri, "Discovering Expressive Process Models by Clustering Log Traces," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, 2006.
- [23] V. Hodge and J. Austin, "A Survey of Outlier Detection Methodologies," *Artificial Intelligence Rev.*, vol. 22, no. 2, pp. 85-126, 2004.
- [24] E. Vidal, F. Thollard, C. Higuera, F. Casacuberta, and R. Carrasco, "Probabilistic Finite State Machines—Part II," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, July 2005.
- [25] G. Ammons, R. Bodik, and J.R. Larus, "Mining Specifications," *SIGPLAN Notices*, vol. 37, no. 1, 2002.
- [26] G. Greco, A. Guzzo, and G. Manco, "Mining and Reasoning on Workflows," *IEEE Trans. Knowledge and Data Eng.*, vol. 17, no. 4, Apr. 2005.
- [27] J.E. Cook and A.L. Wolf, "Event-Based Detection of Concurrency," *Proc. SIGSOFT Foundations of Software Eng. (FSE)*, 1998.
- [28] Process-Mining-Group, "ProM Workflow Mining Prototype," <http://is.tn.tue.nl/research/processmining/tools.htm>, 2006.
- [29] M. Hammori, J. Herbst, and N. Kleiner, "Interactive Workflow Mining: Requirements, Concepts and Implementation," *Data Knowledge and Eng.*, vol. 56, no. 1, pp. 41-63, 2006.
- [30] G. Ammons, D. Mandelin, R. Bodik, and J. Larus, "Debugging Temporal Specifications with Concept Analysis," *Proc. Conf. Programming Language Design and Implementation (PLDI)*, 2003.
- [31] T.W. Malone et al., "Tools for Inventing Organizations: Toward a Handbook of Organizational Processes," *Management Science*, vol. 45, no. 3, 1999.



**Hamid R. Motahari-Nezhad** received the BSc degree in computer science from the Ferdowsi University of Mashhad, Iran, and the MSc degree in computer science from the Amirkabir University of Technology, Iran. He is a PhD student in the Service Oriented Computing Group, University of New South Wales, Australia. His research interests include Web services interactions analysis and management and service engineering. He is a member of the

IEEE and the Australian Computer Society.



**Régis Saint-Paul** received the MSc and PhD degrees in computer science from the University of Nantes, France. He is a researcher at CREATE-NET, a research center in Trento, Italy. Previously, he spent two years as a research associate in the Service Oriented Computing Group, University of New South Wales, Australia. His research interests include service-oriented architectures, database systems, data mining, data summarization, and

end-user programming. He is a member of the IEEE Computer Society and the ACM.



**Boualem Benatallah** is a professor at the University of New South Wales, Sydney, where he is the founder and the leader of Service Oriented Computing Group. His research interests include Web service protocols analysis and management, enterprise services integration, process modeling, and service-oriented architectures for pervasive computing. He has published widely in international journals and conference proceedings, including the *IEEE Transactions on Knowledge and Data Engineering*, the *IEEE Transactions on Software Engineering*, *IEEE Internet Computing*, *IEEE Intelligent Systems*, and the *VLDB Journal* and in the IEEE ICDE, IEEE ICDS, WWW, and ER conferences. He is a member of the IEEE.



**Fabio Casati** is a professor at the University of Trento, Italy. Previously, he was a senior researcher at Hewlett-Packard, Palo Alto, California. His research interests go into three main directions: the first is on middleware for integration and in the analysis of middleware data for improving the integration. The second is about bringing and extending traditional integration technologies to all enterprise data and to the Web. The third, and most important thread, is

related to improving how scientists produce, disseminate, evaluate, and consume scientific knowledge.

▷ **For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/publications/dlib](http://www.computer.org/publications/dlib).**