A Framework for Protocol Discovery from Real-World Service Conversation Logs

Hamid Reza Motahari-Nezhad, Régis Saint-Paul, Boualem Benatallah
School of Computer Science and Engineering
The University of New South Wales
Sydney, Australia
{hamidm|regiss|boualem}@cse.unsw.edu.au

Fabio Casati
University of Trento
Trento, Italy
casati@dit.unitn.it

TECHNICAL REPORT
UNSW-CSE-TR-0623

December 2006

Last updated: July 2007

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 functionality that simplifies the service development lifecycle. It also greatly facilitates the monitoring, visualization, and aggregation of interaction data.

This paper presents a framework for discovering protocol definitions from real-world service interaction logs. It first describes the challenges in protocol discovery in such a context. Then, it presents a novel discovery approach, which is widely applicable, robust to different kinds of imperfections often present in real-world service logs, and helps to derive protocols of small sizes, thanks to heuristics. As finding the most precise and the smallest model is algorithmically not feasible from imperfect service logs, finally, the paper presents an approach to refine the discovered protocol via user interaction, to compensate for possible imprecision introduced in the discovered model. The approach has been implemented and experimental results show its viability on both synthetic and real-world datasets.

1 Introduction and Motivations

A trend that is gathering momentum in Web services is to include as part of the service description not only the service interface (WSDL) 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 [4, 8]. A *conversation* consists of a sequence of messages exchanged between two or more services to achieve a certain goal, for example to order and pay for goods.

Modeling business protocols brings several benefits to Web services: (i) it provides developers with information on how to program clients that can correctly interact with a service; (ii) 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; (iii) 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 [8]; (iv) 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). Following our previous work [8], we adopt final 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 Figure 3).

This paper investigates the problem of discovering protocol models by analyzing real-world service conversation logs [30, 31]. 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; Even when the protocol information is documented, like in Google Checkout service[1], the protocol specification need to be extracted from the textual description, which is not as precise as a formal model. In addition, the documentation may not be consistent or has not been kept updated with the service implementation.

2. *Discovering cross-service protocols*: To be able to analyze a set of Web services, sometimes we are interested in finding protocols that span across Web services. For example, in a typical ordering procedure, several web services of the client communicate with several web services of the seller (e.g.,

[1] code.google.com/apis/checkout/developer/index.html

there are quotation services, ordering services, payment services, etc). Even if protocol specifications are available for each individual services, what is important here is to understand the protocol model governing the overall interaction among the partners, for example as that can be taken as the basis for monitoring and analyzing executions or because it may lead the provider to understand the specifications of what can be a single, unified order management Web service.

3. *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 [2]. Finally, as the implementation of service evolves, the protocol definitions become increasingly incorrect. Automated protocol discovery helps in maintaining a correct and up-to-date protocol definition.

1.1 In Search of a Protocol Discovery Problem Statement

Ideally, if we had a *perfect* log, i.e. a log that includes all possible conversations modulo loop (*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 problem could be defined 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 smaller one is generally preferable as it easier to understand for users. For example, if state machines are used, the smaller size model is the one with a smaller number of states. The smallest possible model has only one state, called the *flower model*. For instance, Figure 1(a) shows the flower model for messages a, b, c, d. However, it is likely to be over-generalized as it accepts any conversations consisting of a, b, c, d some may not be present in the log, and so it is not useful. On the other hand, it is possible to build a state machine that has a path for each distinct conversation in the log, called the *glove model* (or the *prefix tree*). Such a model would be so complex having too many states. It is likely to be an overfit: it is too precise since it accepts only the conversations in the log. However, the set of supported conversations is infinite given that protocols allow loops. So, this model is not useful either. It has been shown that finding the minimal model that fulfill the requirement of above problem is undecidable [20,21], nevertheless, probabilistic and algorithmic approaches for finding an approximate for the minimal model exist [33].

However, the problem formulated as such is unrealistic for real-world service logs. As widely observed [24, 29, 34, 36, 42], real-world logs are imperfect, i.e. both incomplete and noisy (e.g., do not record some messages). Experiments with commercial logging tools, e.g., HP SOA Manager, also confirm that it frequently happens 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. The impact of noise on complexity is intuitive:

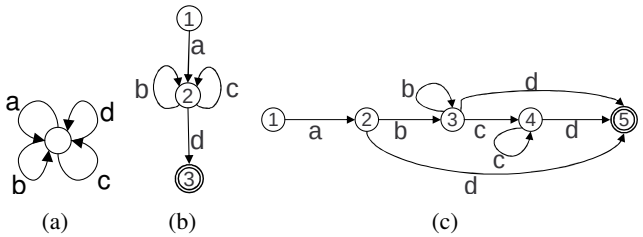


Figure 1: (a) flowermodel protocol for messages a, b, c, d, (b) FSM to be discovered, (c) the same FSM in absence of some conversations

actual protocol model are typically fairly simple, and hence allow a limited set of conversations (modulo loops). However, when noise is present (e.g., messages may be logged in the wrong temporal order, or may be missing), even a small level of noise may create over time a large number of conversations hence leading to a more complex model. The model we obtain from an imperfect log is not only inaccurate, but may be very complex.

The issue for incompleteness is more subtle but equally problematic. Consider the service protocol depicted in Figure 1(b). Conversations allowed by this protocol include *ad*, *abcd*, *abd*, *acbd* and *acd*. If the log does not contain conversations such as *acbd* (and generally $ac + b + d$), the discovery algorithm will infer the protocol depicted in Figure 1(c). This protocol conforms with the conversations found in the log, i.e. it correctly accepts all the logged conversations and rejects the conversation *acbd*. Such a precision, however, leads to discovering a more complex protocol (i.e. has more states and transitions) than the real protocol.

The above observations show that proposing a solution for discovering protocols from imperfect logs is not trivial. 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 proposing a framework, algorithms and tools to support the discovery of protocol models from imperfect service logs. We focus on addressing the above mentioned problems in a three-stage approach depicted in Figure 2.

Protocol Discovery Framework. We characterize the different dimensions of the problem of protocol discovery for Web services. We describe and discuss the different conceptual components that need to be part of every solution to discover protocol models from real-world service message logs.

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 [3, 12, 38]. 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 [3, 12, 38]. We propose in Section 4 a quantitative measure and an

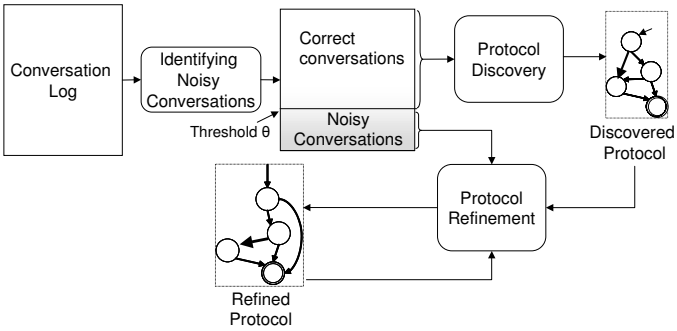


Figure 2: Stages of the proposed protocol discovery approach

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, different from each other (as it is unlikely that the same error occurs in the same place), and also often different from those usually produced by the service. This means that many of those noisy sequences will have a 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. In our experiment, this approach led to correctly identify noisy conversations with a precision above 90% in logs containing from 0 to 30% of noise (Section 7).

Protocol Discovery. We propose a protocol discovery algorithm that aims at discovering a simple (small-sized), precise model, and that can cope with the problem caused by log incompleteness mentioned above. The proposed algorithm first constructs a over-generalized, simple model of the target protocol, by analyzing message sequences in the log. Then, it progressively refines and extends the initial model to ensure that conversations considered noisy or not present in the log are not accepted by the discovered model to improve the precision of discovered model (See section 5).

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 it becomes known, e.g., through statistical analysis, that a given operation can be invoked at any time, e.g., the operation *CheckBalance* in a shopping service, it is possible to infer additional correct conversations that are not recorded in log. Doing so, our approach allows discovering simpler models, which are closer to the designed protocol models. To the best of our knowledge, no existing model discovery approach considers the issue of log incompleteness.

Protocol Refinement. Log imperfections makes it in general impossible for an automated approach to always derive simple, precise and correct protocol mod-

els. We propose a novel approach to *visual, interactive and user driven protocol refinement* (Section 6) to compensate for errors made during discovery step. User-driven refinement is a challenging task: it is unrealistic to ask users to go through all conversations in the log, inspect them and refine the protocol to correct possible mistakes because conversation logs can be very large in size. We propose two mechanism to make the protocol refinement simple and effective (Figure 2):

- We augment discovered protocol with various meta-data including (final) state and transition support to represent uncertainty due to log imperfection. The purpose of this refinement is to improve the precision of discovered protocols by updating the discovered model to exclude conversations that should not have been accepted. The metadata help in identifying incorrect conversations that are mistakenly accepted by the discovered model.

- We present an approach for analyzing all conversations that are in the log, but can not be accepted by the discovered model, to compensate for mistakes due to exclusion of correct but infrequent conversations. This approach is based on defining: (1) a notion of *distance* between protocol model and excluded conversations, and (2) *manipulation operations* that make modifications in the model so it accepts these conversations. We show that the refinement process can be facilitated by having users analyze manipulation operations rather than conversations.This refinement helps in improving the recall of the discovered model, i.e. the percentage of correct conversations that are accepted by the model.

Protocol Discovery Tool. The final contribution of this paper is implementing the proposed approach in a tool and validating the approach via experiments performed actual service execution logs (Section 7). The tool is part of ServiceMosaic project [8, 31], a platform for model-driven analysis and management of Web service protocols.

The paper is structured as follows. In Section 2 we give basic definitions and characterize imperfections in service logs. In Section 3 we propose a framework consisting of different components, which are required in any protocol discovery solution. Section 4 presents our approach for identifying noisy conversations. In Section 5, we present the protocol discovery algorithm. The proposed protocol refinement approach is explained in Section 6. Implementation and evaluation of the proposed approaches are discussed in Section 7. Related work is presented in Section 8. We conclude and introduce areas of future work in Section 9.

2 Preliminaries and Assumptions

2.1 Modeling Business Protocols

Following our previous works [8], we choose to model business protocols as deterministic Finite State Machines (FSM) as it is a well-known paradigm with es-

tablished formal foundations, and it is simple and suitable for modeling reactive behaviors. 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, $T \subseteq S^2 \times M$ 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* (resp. *reject*) the sequence of messages of a conversation c if a path (necessarily unique) exists (resp. doesn't exist) from the initial state to one of the final states.

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 Figure 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.*

Note that in general we could choose a finer-grained protocol model where transitions are not only associated to messages, but also to conditions on message parameters. For example, we may want to specify that a `ShippingInfo` message where parameter `shipType` has a value of *ground* may lead to a different state than when `shipType` has a value of *air*. We will briefly touch upon the implication that this finer-grain model brings to protocol discovery later in this section.

2.2 Service conversation logs

We assume that service (message) logs contain the following information: (i) message transfer information (sender, receiver, timestamp), (ii) message name, and (iii) 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

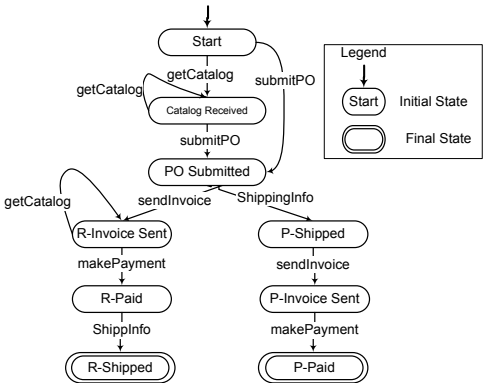


Figure 3: The business protocol of the `Retailer` service

service monitoring tool [16, 34]. 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. For example, if standards such as WS-Coordination or WS-Conversation are used, conversation IDs can be extracted from SOAP headers of messages. The difficulty of this task can range from trivial, as in the above example, to challenging (e.g., when information to use, and how to use them, are unknown and have to be inferred from the log). The problem of mapping messages to conversations constitutes an independent research thread in its own right [32].

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* τ *is the timestamp.*

The above definition is introduced to closely model real message logs, which indeed are tables of entries. However, in the context of protocol discovery, it is more handy to consider conversations as the basic piece of data. Hence, in what remains, we will consider conversation logs as our input dataset. 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* $CL = \{c_1, c_2, \ldots, c_n\}$*. Each conversation* $c_i \in CL$ *is a sequence of messages* $c_i = \langle m_1^i, m_2^i, \ldots m_k^i \rangle$*. Conversations in CL are obtained from a message log by grouping entries per conversation and by ordering them according to their timestamp.*

A conversation c can thus be seen as a sequence of symbols—a string—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 crash).

- *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 *a***bc***de*, we may find *a***cb***de* 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 timestamps of messages or performance degradation so both *b* and *c* get the same timestamps, e.g., we have experienced this using HP SOA Manager [1] due to the performance reason. Also if conversations are logged on more than one servers, it can be difficult to establish the correct ordering between messages of the same conversation [34].

- *Partial conversations* – We call *partial* a conversation that is interrupted before its completion. This can be due, e.g., to network failure, client abortion or service execution exceptions. For instance, if *abcd* is the message sequence of 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 of Figure 1(c), state 5 is marked final which indicates that *abcd* is complete. The presence of a 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 Figure 1. We present an approach for handling incompleteness in service log in Section 5.3.

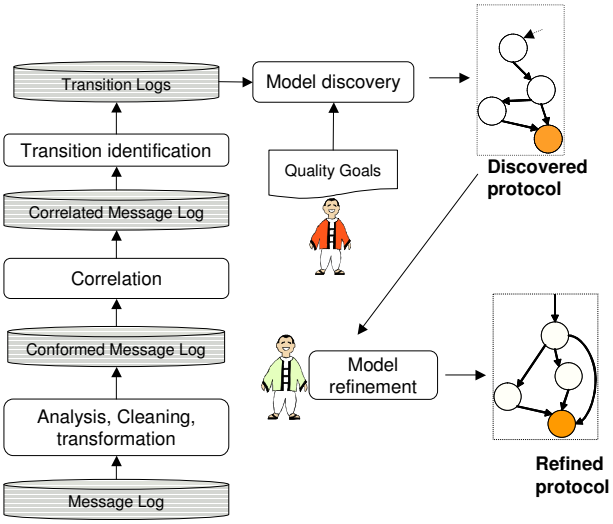


Figure 4: Phases of Protocol Discovery

3 The Protocol Discovery Framework

This section presents the framework we propose for protocol discovery, in which we identify conceptual components that should be ingredients of any protocol discovery solution. Then, we specify the scope of the solution presented in this paper.

3.1 Phases of the protocol discovery process

The protocol discovery problem has many facets. These facets correspond to the steps (represented as rounded boxes) of Figure 4, and are briefly described below. Different solutions can be devised to deal with the problems posed by each of these steps. However, unless realistic assumptions can be made for why some of these steps are not needed or not applicable in a certain discovery scenario, an approach to tackle each of them must be devised.

Log analysis, cleaning and transformation. This step consists in collecting data from the service interaction logs (also called *message logs*) and in transforming them into the format supported by the discovery tool. In most cases, a message log is essentially a repository that contains the messages sent or received by the services of a company (typically monitored by the company itself). In devising a generic protocol discovery solution, it is also important to make reasonable assumptions on the kind of information present in the logs and verify if these assumptions are satisfied by the log at hand. An important aspect in log analysis is to measure noise level (i.e. the ratio of incorrect conversations with respect to the total number of conversations) that is introduced by the logging system. Different log systems are likely to introduce different kinds of noise and imprecisions, and

so should be considered in the solution. The output of this step is called *conformed message log*, which include messages as well as meta-data about estimated noise types and levels.

Correlation. Tools that monitor message exchanges among services provide logs that are a set of messages. To derive protocol models from message logs, we first need to correlate messages into complete conversations. Once we have conversations, and hence the different paths that have been taken through the protocols, we can derive the protocol model from this set of paths. In some cases, messages in the log contain *conversation identifiers*, which allow to identify each exchanged message belong to which conversation. Unfortunately, this is rarely the case in real applications, and even in those rare cases, this would not help if we are interested in interactions among *business* services. Therefore, to prepare the data for model discovery, a mechanism to map messages to the conversation they belong to must be identified. The output of this step is called *correlated message log*, or a *conversation log*.

It is possible to devise an algorithm to analyze the different messages and determine that certain message elements are important for conversation correlation. For example, to finds out that *product number*, *customer ID*, and *purchase date* identify messages belonging to the same conversation occurring between a customer and a purchase order submission service, and further that *product number*, and *purchase date* could help in identifying the correlation of messages between the PO service and the shipper (in BPEL terminology, this is analogous to discovering *correlation sets*). We have investigated this problem and proposed a solution for identifying different ways of grouping messages in a log into conversations, which is presented in [32].

Transition identification. As mentioned before, transitions between states of a protocol can depend not only on messages, but also on message parameters. In fact, transitions in the FSM can be associated not only to the message type, but also to the content. For example, a message response with parameter *approval=yes* may lead to a different state than the same message with parameter *approval=no*. Hence, part of the protocol discovery problem consists in discovering which parameters of which messages (and which values for these parameters) correspond to different transitions. Furthermore, a protocol model may also allow timed-controlled transitions or other sophisticated modeling constructs [8]. This means that by looking at a message log and in particular if we limit the analysis to message names and not to their parameters, we cannot determine which transition each message corresponds to (and specifically, if two messages with the same name but different parameters correspond to the same or to different transitions). Hence, part of the protocol discovery problem lies in analyzing the message logs and in mapping each log entry into a transition, and in also adding possible time-controlled transitions if these are allowed by the protocol model. The output of this step is called *correlated transition log*. At this stage, we have a log that contains conversations composed of an ordered set of transitions, and we are ready to derive the protocol model.

Protocol discovery algorithm. This is the heart of the protocol discovery

problem, and it includes several aspects that must be considered. First, there is the need of defining goals and quality criteria for the discovered model. In general, deriving a model that can generate the conversations in the log is easy. For example, we could generate a FSM that has a path for any distinct conversation found in the logs. This would lead to a protocol that has a huge number of states and transitions. Alternatively, one could "discover" a *flower* model, which has one state and a self-transition for each distinct message name in the log. This model is very simple and can generate all conversations in the log, but it can also generate any conversations of any length given that set of messages, which means that the model is not really providing any useful information. The second aspect is related to how to take into account incompleteness in the logs. Finally, once the characteristics of the input and the goal of the output are clear, the algorithm must identify how to derive high-quality protocol models from potentially noisy logs.

Protocol refinement. The fact that logs can be noisy and incomplete makes it impossible in the general case to discover the exact protocol model supported by a service. The protocol discovery algorithm is supposed to take care of doing whatever can be done automatically in terms of discovering the best possible protocol model based on the defined quality criteria, but once a model has been derived, there is the need of assisting users in refining and correcting the protocol model, based on knowledge or hypothesis that users may have for some parts of the protocol. The challenge here consists in how to measure and rank the possible errors in the protocol model and in how to progressively guide users through them in a way that is as simple and as effective as possible, so that corrections can be provided and the protocol can be refined.

Note that sometimes the above steps can be repeated or executed in a different order with respect to what presented above. For example, users may be able to provide knowledge they have on parts of the protocol even before the algorithm is run, to better guide the algorithm. We also believe that, although the above discussion focuses on protocol discovery, analogous if not identical steps have to be performed for other model discovery problems, including for example process discovery from business process execution logs.

3.2 Scope of the present work

In this paper, we focus mainly on three steps of the above framework: noise analysis in conversation logs, protocol discovery and protocol refinement (see Figure 2). Hence, we assume that correlation information is available in the logs (we have the conversation log) and that different message names correspond to different transitions. We do not discover timed transitions or transitions that include conditions based on message parameters. In other words, we assume that the message log as written by the logging application is in fact already a correlated transition log. We have investigated the problem of message correlation and in [32] proposed a interactive and visual approach for correlation of messages in message logs into

conversation logs, and we leave transition identification as future work.

4 Identification of Noisy Conversations

4.1 Characterizing noise distribution

As discussed in Section 1.1, 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. We assume that errors made by logging systems can be modeled as a Poisson process [28]. In the vocabulary of log, a Poisson process is defined by the following properties:

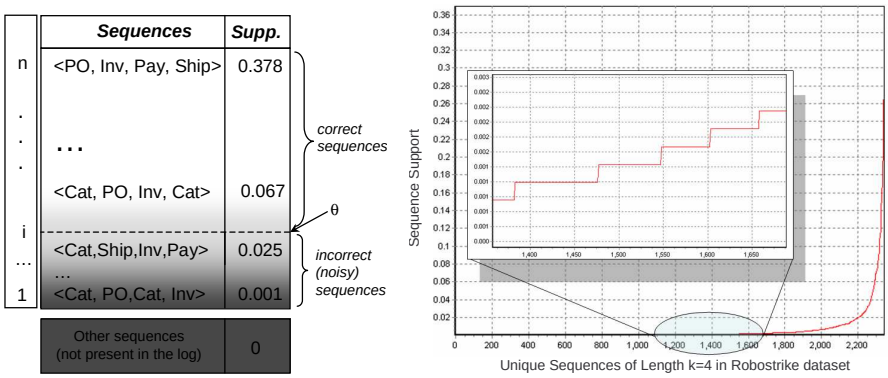
- The number of errors in non-overlapping sequences of log entries is independent for all sequences;
- The probability of exactly one error on a given sequence is proportional to the sequence length;
- The probability of two or more errors in a sufficiently small sequence is essentially null.

The above properties, defined for a message log, are thus assumed to be also valid when considering sequences of messages of a same conversation.

Note that the above assumption is not always verified: Errors found in service logs may depend on the current state of the service being logged. For example, a service operation may require more computing resources than others, impacting the logging system performance and resulting in an increased error rate for these operations. If the dependency between messages and errors is too important, our approach will fail to identify the resulting conversations as noisy. Indeed, the more deterministic is an error and the more difficult it is to differentiate it from the normal behavior of a service. For example, if messages a and b are almost always swapped by the logging system (i.e. although it occurs before, a is almost always timestamped after b), the protocol will have all the appearance of being ba rather than the correct ab. 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 [42].

4.2 Filtering Noisy Conversations

Coming back to the above assumptions, a first observation is that the longer a conversation, the more likely it contains an error. This implies that short sequences are more robust to noise than long ones. For this reason, it is preferable to base protocol discovery on sequences of short length, say of length k, rather than on conversations. An additional merit of this approach is that even very long conversations, which are likely to be noisy, consist of several subsequences, most of



(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

Figure 5: The analysis of logs

them correct. Thus, noisy conversations, too, can contribute to the identification of correct sequences.

For the size of the sequences, we have learned through experiments (see Section 7) 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 dataset in hand (See Section 7). 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 4.1. *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* $Q^3 = \{\langle abc \rangle, \langle bcd \rangle, \langle acb \rangle, \langle abc \rangle, \langle cbd \rangle\}$.

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 θ_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 $(q_1, q_2, \ldots, q_n)$ such that $\forall i < n, supp(q_i) \leq supp(q_{i+1})$ and $q_i \in Q^k$ (Figure 5(a)). Figure 5(b) represents the histogram of sequence support for the game service conversation log (details of this dataset is presented in Section 7).

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 which have the same support of the previous step point). Consider now the ratio γ_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 rapidly decreasing with m since 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 $\theta_k = supp(q_{m_0})$. This approach works well in different real datasets, besides being consistent with the intuition. In addition, it is "robust" as the index m for which $\gamma_m < 0.5$ or $\gamma_m < 2$ is about the same (the difference concerns only a small number of sequences when compared with the total number).

4.3 Time complexity of the noise identification

The size of the histogram directly 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 operation. In practice, each operation is followed by only a subset of operations. If ϕ 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 Section 7), with a protocol of 32 operations, less than 2400 unique sequences of length 4 were observed (Figure 5(b)), among 25000 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 in Section 7). Finally, the number of unique sequences is independent of the size of the dataset (the number of conversations in *CL*), i.e., new sequences are found less and less frequently as the dataset grows. Hence, the time complexity of the noise identification depends linearly on the number of conversations (as we parse the data only once).

5 Protocol Discovery Algorithm

Let θ_k to be the noise thresholds for sequences of length k estimated using the approach presented in Section 4. We denote by $Q^k_\theta = \{q \in Q^k | supp(q) \geq \theta_k\}$, the

Algorithm 1 The Protocol Discovery Algorithm

Require: k (the sequence length), Q_θ^k (the set of correct sequences)
Ensure: $DP = (S, s_0, F, M, T)$
1: Construction of an initial message graph $G_{initial}$ from Q_θ^k
2: Enhance precision of $G_{initial}$ based on Q_θ^i, $3 \leq i \leq k$
3: Convert $G_{enhanced}$ to DP
4: Minimize DP

set of correct sequences of length k, i.e. the set of sequences which have a support greater than θ_k. The set of unique sequences of length k of a given conversations c is defined as $c^k = \{q \in c || q| = k\}$. Then, we revisit the problem of protocol discovery as the following:

Problem 5.1. *The protocol discovery problem is to find the minimal discovered protocol (DP) (minimal deterministic finite state machine) that:*

(i) DP 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 c for which all subsequences of length k of them* (c^k) *are found in* Q_θ^k.

(ii) DP rejects all conversations $c \notin CC_k$.

This problem is a variation of the well-known problem of regular grammar inference from sample input [33], and it has been proven that there is no efficient algorithm for finding the minimal *DP* [20,21]. 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 which 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 [33]. 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 [33].

It has been shown that software interaction models are much closer to generalized models than to specialized ones [26]. Several work in software interaction model discovery take such an approach and start from a simple, over-generalized model, e.g. [12,14,26]. We have also adopted this approach for protocol discovery.

5.1 Overview of Protocol Discovery Algorithm

Algorithm 1 shows the protocol discovery steps. The inputs of the algorithm are parameter k and the set of correct sequences Q_θ^k. Its output is a deterministic FSM representing the discovered protocol (*DP*). Algorithm 1 has the following four steps:

Step 1: An initial message graph ($G_{initial}$) is constructed from the set of correct sequences Q_θ^k. It has one node for each unique message in the log, and directed edges between nodes are established based on Q_θ^k;

Step 2: $G_{initial}$ is overgeneralized, i.e. its precision is low, in the sense that it accepts some sequences that are not in Q_θ^k. The lack of precision 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. To enhance the precision, separate paths for correct sequences are created to prevent acceptance of incorrect ones;

Step 3: In a final 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;

Step 4: Finally, the obtained FSM may contain some redundancy in form of equivalent states, i.e. states with the same output transitions going to the same target states. Equivalent states can be merged to reduce the size of FSM. The result of this step is the discovered protocol *DP*.

Step 1 of the algorithm is performed in a similar way as in the approach of Cook et. al. [11,12] which is presented for discovering software workflow models represented as non-deterministic FSMs. In step 2 a new approach for specialization of the initial model is presented to cater for the requirements of protocols, which are represented as deterministic FSM (Section 5.2). In step 4 we use algorithmic minimization techniques to reduce the size of *DP*. In addition, we present a novel approach for handling log incompleteness to simplify *DP* to make it similar to the real-world representation of *DP* (Section 5.3).

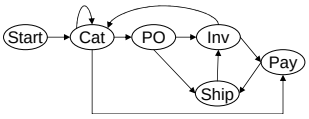
5.2 Protocol Discovery Steps

5.2.1 Construction of a message graph

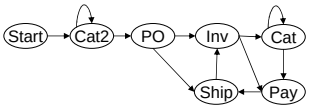
In this step, a generalized representation of *DP* is built as a directed graph called $G_{initial}$. We consider initially one node in $G_{initial}$ for each unique message in the log. For instance, 5 nodes are created in $G_{initial}$ corresponding to `Cat`,`PO`,`Inv`,`Pay`, and `Ship`. Then all sequences $q \in Q_\theta^k$ are used to connect these nodes in $G_{initial}$. Assuming that $k = 4$, for sequence $q_1 = \langle$`Cat,PO,Inv,Pay`$\rangle$ a directed edge from node `Cat` to node `PO`, `PO` to `Inv`, etc is created. Figure 6(a) shows the $G_{initial}$ built from the log of the `Retailer` service shown in Figure 3 using Q_θ^4. Note that there is no transition from the node `Start` to `PO` as the support of sequences with such transition is below θ_4.

5.2.2 Graph precision enhancement

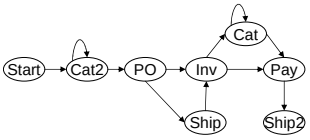
The graph $G_{initial}$ is often *overgeneralized*: it accepts, by construction, all sequences of Q_θ^k, but also some sequences which are not in Q_θ^k. We start checking for such sequences from the smallest length, i.e. $i = 3$ to k progressively. For example, for $i = 3$, $G_{initial}$ accepts sequence $\langle$`Start`,`Cat`,`Pay`$\rangle$ which is not in Q_θ^k, and so it is an incorrect sequence. We call 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.



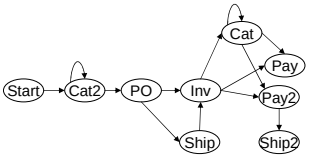
(a) The initial graph $G_{initial}$ for `Retailer` dataset



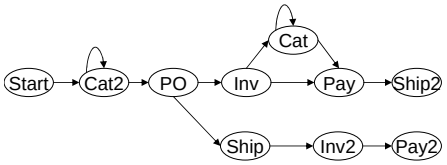
(b) $G_{initial,1}$ after splitting node *Cat* in $G_{initial}$ for $k = 3$



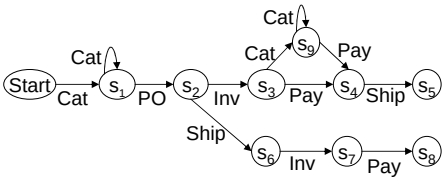
(c) $G_{initial,2}$ after splitting *Ship* node for $k = 3$



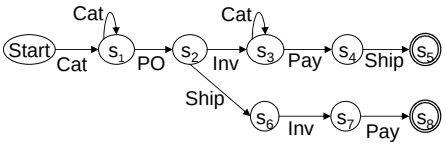
(d) Splitting only the node next to the last (*Pay*) in $G_{enhanced,2}$



(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

Figure 6: Applying the discovery algorithm steps on `Retailer` service log

Algorithm 2 The Precision Enhancement Algorithm

Require: Initial Graph $G_{initial}$
Ensure: Enhanced Graph $G_{enhanced}$
for $i = 3..k$ **do**
  for each node v of $G_{initial}$ **do**
    $IS \leftarrow IS \cup il(v,i)$
  end for
  $prefix \leftarrow$ all unique $seq[1..(i-1)]$ in IS
  for each $seq \in prefix$ **do**
    $IE \leftarrow$ all v', $(prefix, v') \in IS$
    for each node $seq[j]$ in $seq[2..(i-1)]$, $2 \leq j \leq (i-1)$ **do**
      $seq[j]' \leftarrow$ copy of $seq[j]$ in G
    end for
    for each node $seq[j]$ in $seq[2..(i-1)]$, $2 \leq j \leq (i-1)$ **do**
      if $j > 2$ **then**
        create an edge from $seq[j-1]'$ to $seq[j]'$ in G
      end if
      Copy outgoing edges of $seq[j]$ to $seq[j]'$ except the edge from $seq[j]$ to $seq[j+1]$
    end for
    copy outgoing edges of node $seq[i-1]$ to node $seq[i-1]'$ except for edges to nodes in IE
    remove edge $(seq[1], seq[2])$
    add edge $(seq[1], (seq[2])')$
    update IS
  end for
  remove all nodes not reachable from `Start`
end for

To address this issue, we propose Algorithm 2 for enhancing the precision of $G_{initial}$. For each incorrect sequence of length k a new path in the graph is created to only allow correct sequences to be accepted. The creation of the new path is done by copying all the middle nodes of the incorrect sequence, i.e. nodes $2..k-1$ (e.g., second and third nodes in a sequence of length 4), denoted by $2'...(k-1)'$. 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 acceptance of the incorrect sequence.

To demonstrate how Algorithm 2 works, we apply it on $G_{initial}$. First, let's consider the above incorrect sequence. 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`. Then, the edge from `Start` to `Cat` is removed and an edge from `Start` to `Cat2` is created. The result graph is called $G_{initial,1}$. It does accept the incorrect sequence <Start,Cat,Pay> (Figure 6(b)). The next incorrect sequence of length 3 is ⟨Pay,Ship,Inv⟩ which is handled in a similar way by copying `Ship` node. The result graph, called $G_{initial,2}$, is depicted in Figure 6(c). Then, we check for incorrect sequences of the next higher length $i \leq k$. Here, $i = k = 4$. The sequence ⟨Ship, Inv,Pay,Ship2⟩ 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 is 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 not possible to accept the incorrect sequence anymore (see $G_{initial,3}$ (also called $G_{enhanced}$) in Figure 6(e)).

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 splitting (and so copying) 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 ith 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.

It should be noted that a splitting method for enhancing graphs is also proposed by Cook et. al. in [11, 12] which recommends splitting the node next to the last node in an incorrect sequence. However, this could not always be applied to deterministic FSM of service protocols, as it does not allow for removal of the incorrect sequence when more than one nodes before the last are shared between correct and incorrect sequences. For example, assume the graph of $G_{initial,2}$ and the incorrect sequence ⟨Ship1, Inv,Pay,Ship2⟩. Splitting only the node before the last, i.e., Pay, results in the graph shown in Figure 6(d). 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 non-determinism after transformation of $G_{enhanced}$ to FSM. A non-deterministic FSM is not desired for service protocols.

5.2.3 Conversion into a FSM

We convert the enhanced directed graph $G_{enhanced}$ (Figure 6(e)) to a FSM by naming transitions to the name of their target nodes. The resulting graph is a deterministic FSM (Figure 6(f)).

5.2.4 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: (i) the graph constructed in step 1 may not be the minimal form of *DP*, and (ii) 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. For example, in Figure 6(f), states s_3 and s_9 are equivalent. Hence, in this step we

minimize the FSM without changing the set of conversations it accepts. We use the Ullman-Hopcroft minimization algorithm [27] for this purpose. Figure 6(g) shows the minimized discovered protocol for the `Retailer` service. The final states in FSM are identified by using all the conversation in the log to specify the states, in which most of conversations terminate.

5.3 Handling Log Incompleteness

Despite the minimization, experiments (see Section 7) revealed 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 (e.g., operation *CheckBalance* in the shopping service). These are very common in protocols, and can be nicely 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 non-occurrence of the operation indeed 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 minimize states, while pervasiveness helps us 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 very cluttered with arcs and hence hard to read. Hence, failing to recognize transparency and pervasiveness due to incomplete logs has a big impact on the quality of the discovered model.

To handle this issue, we propose the following heuristics: we look for transparent and pervasive operations in the initial directed graph G, in which each node correspond to an operation, in two steps. In the first step, we identify nodes in G 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 (i.e. $\forall p \in Pred(o)$, there is an edge from p to o in G), 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), 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 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. So, 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% 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. After discovering all transparent operations o and their corresponding predecessors p, we remove the edges from $p \in Pred(o)$ to o in G. Then we apply steps 2 and 3 of the algorithm (splitting and conversion to FSM) on G. Finally, we put back edges that we removed, as self-transitions labelled 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 G 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.

5.4 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_θ^k. This is done in a single pass through the set Q_θ^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 4 the notation ϕ 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_θ^k). Thus, the average arity of each node in $G_{initial}$ is less than ϕ. We denote by φ ($\varphi \leq \phi$) 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 ($Q_{\theta_i}^i$). Generating the sequences is done by traversing the graph and, for each node, by performing a depth first search stopping at a depth i from the node (when possible).

A total of $p\varphi^{i-1}$ sequences are thus generated and looked up in the list Q_θ^k steps at most which is also the number of sequences to be checked. For checking if sequences are in the list $Q_{\theta_i}^i$, the lookup operation performs very fast since sequences in $Q_{\theta_i}^i$ 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. We mentioned this step so as to ease the description of the overall algorithm, but the graph can be maintained in memory in a similar way from the beginning, associating labels with edges rather than nodes.

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 [27]. 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 δ 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_θ^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 δ is necessarily small. This is natural since the initial graph was connected using precisely the set of correct sequences Q_θ^k. Estimating more precisely the number of split operations would require to introduce additional criteria on the dataset which would not be very intuitive. In our experiments (see Section 7), the size of the FSM before minimization was comprised between 100 and 200, which corresponds to between 26 to 60 split operations. 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 ϕ of operations that follow a given operation. However, we can see that the size of the dataset do not impact the time complexity of the algorithm, but only that of the noise identification algorithm presented in section 4. Experiments in Section 7 show that the algorithm is practically scalable with regard to k, p and ϕ in real datasets.

6 User-Assisted Protocol Refinement

6.1 Guiding principles

There are two main reasons why users may require to refine the protocol discovered after the previous phase: First, 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 by mistake identified as noisy. This happens because we do not have a priori knowledge of which conversations are noisy and hence we need to estimate a noise threshold. Second, depending on the intended usage of the discovered protocol, the user may wish to simplify the model. For example, there may be conversations that, although possible, are not interesting from a monitoring perspective and hence the user may wish to remove the corresponding part of the FSM from the model.

We assume the second case could be handled by providing a graphical editor

and operations to allow removing states and transitions (See Section 7.6). Handling the first case is instead interesting and challenging for two reasons: (i) it is a common situation that occurs whenever there is noise; and (ii) in large logs the number of noisy conversations may be very high, even though noise is random and rare (See Section 4). In practice, it is not uncommon to find thousands of incorrect conversations in the logs and 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 here is how to best help and guide users through the interactive analysis of conversations, which are estimated to be noisy. We call such conversations *uncertain* as we are not sure if they are really noisy or not. To overcome above issues in discovered protocols, we provide interactive refinement mechanisms for discovered protocols by featuring: (1) *meta-data driven protocol refinement*, and (2) *Distance-based interactive protocol refinement*.

6.2 Meta-data driven protocol refinement

The goal of this step is to improve the precision of *DP* by disallowing all conversations that are accepted by mistake. We annotate the discovered protocol model with various meta-data 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). We provide a graphical tool that enables users to visually browse the discovered protocol and examine associated meta-data to understand potential errors made during the discovery process and correct them.

Our tool facilitates the protocol browsing by identifying regions of *DP* that requires more attention in the sense that we are less certain about their correctness. 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). Stronger a line, we are more certain about it in the model, and darker a state, we are more confident it is a final state.

To allow for refining *DP* we provide a set of operations to be applied to *DP*. In particular, we define operations that allows for removing a state, removing a transition, and updating the final state property of states (setting to on/off). Updating the final state property of a state based on discovered meta-data allows for handling partial conversations in the log.

6.3 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 mainly due to setting the noise threshold. In information retrieval terminology, the goal is to improve the recall. 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 4). This is one of the reason that the refinement step should be user-driven to distinguish between these cases.

For above reasons, the problem of distance between a conversation and *DP* is akin to the problem of identifying *edit distance* between strings [40], in which differences between strings are of type of missing, additional and swapped characters. In a nutshell, the distance between an uncertain conversation and *DP* 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, 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, non 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 (i) guiding users through analyzing possible manipulations (small in number) instead of the possible conversations (very high in number), and (ii) ranking manipulations based on how often they occur.

In order to execute the above strategy, we need to: (i) identify the possible operators that can transform conversations to be accepted by *DP*, and (ii) 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.

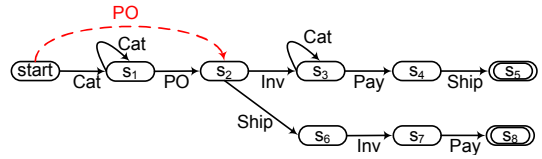
6.3.1 Edit distance and manipulation operators

The notion of edit distance has been studied in the past, mainly by Levenshtein, and here we apply the Levenshtein edit distance [40], which is based on three operations: insertion, substitution and deletion. Specifically, for conversations we consider operations $\texttt{swap}(c, m_1, m_2, s)$, which inverts the order of two messages

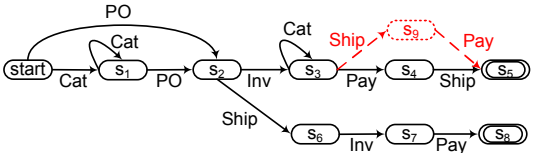
m_1 and m_2 in a conversation string c, where m_1 occurs with the conversation in state s; $\texttt{insert}(c, m_1, m_2, m_x, s)$, which inserts message m_x between m_1 and m_2 in conversation c, again taking the occurrence of m_1 in state s; and $\texttt{delete}(c, m_x, s)$, which removes message m_x 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 most 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 w.r.t. each other, i.e. both ab and ba are possible, however, one of the orders is more common in the log. Insert operation handle the cases where it is believed the logging infrastructure missed logging an operation b in a given sequence ac (so it should have been abc), while ac is 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 6.1. *Consider the protocol DP discovered from the log of the* `Retailer` *service in Figure 6(g), and conversation* ⟨`Po`,`Inv`,`Pay`,`Ship`⟩*. 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. It is not currently supported by DP, although, it is a valid conversation of* `Retailer` *under the designed protocol of* `Retailer` *in Figure 3.*



(a) Insert operation (Cat) in the sequence ⟨Po,Inv,Pay,Ship⟩



(b) Swap operation (*Ship* and *Pay*) operations

Figure 7: Refinement operations on *DP*

6.3.2 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 ⟨Po,Inv,Pay,Ship⟩), we should have an add transition operation that allows adding a transition from the state Start to the state s_2 as in Figure 6(g) (See Figure 7(a)). If the user accepts this change, the transition becomes part of the model (see Figure 9). Refining based on a delete operation translates in adding a transition, too. For example, for conversation ⟨Cat,Po,Inv,Pay,Cat,Ship⟩, 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_4. For a swap operation, we need to add a state and two transitions. For instance, conversation ⟨Cat,Po,Inv,Ship,Pay⟩ suggests a swap between Ship and Pay from state s_3 in Figure 6(g). To refine the protocol based on such swap operation, we need to create a new state, e.g., s_9 and two transitions, one from state s_3 to s_9 for Ship operation, and the other from state s_9 to state s_5 (see Figure 7(b)). These considerations make it clear that we need two refinement primitives on the protocol, one for adding states and the other for adding transitions.

6.3.3 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 (Figure 8). For example, a node can be $swap(.,m_1,m_2,s)$.

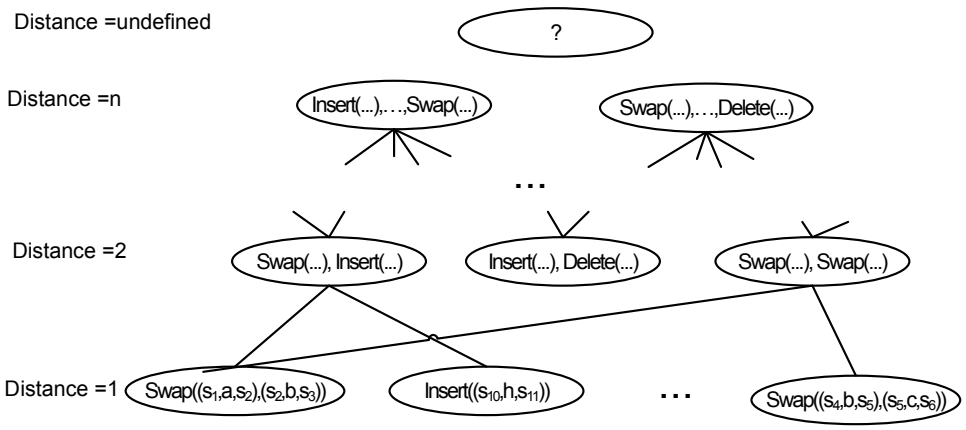


Figure 8: The class hierarchy for classification of uncertain conversations

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 trans-

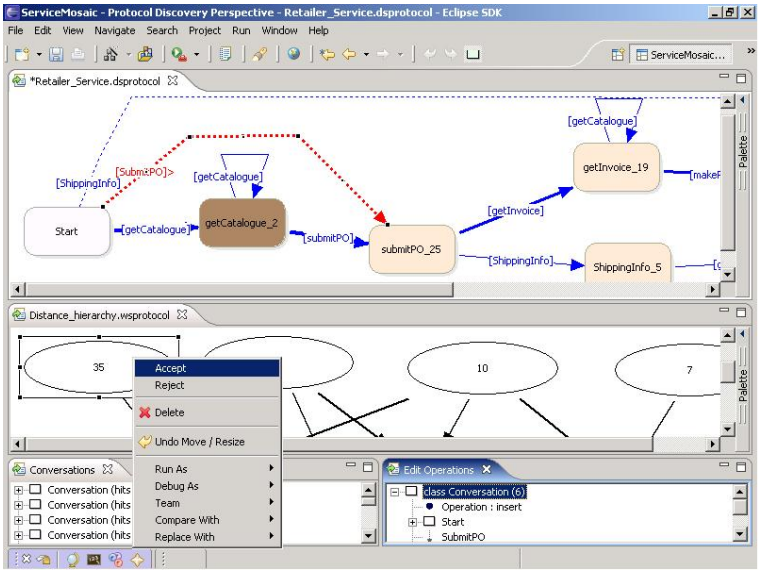


Figure 9: Discovery and Refinement Editor

formed 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 intractive approach that by selecting each node in the hierarchy highlights the refinement operations on *DP* for the user (Figure 9). 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 $\langle$`Po`,`Inv`,`Pay`,`Ship`$\rangle$ allows inclusion of such conversation in *DP*, and it is associated to the highest cardinality node (bottom part of Figure 9). On the other hand, $\langle$`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.

7 Experiments and Validation

7.1 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 then use the classical *recall* and *precision* metrics [5]. Recall is the 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. Then, we utilize precision, recall and the size of discovered protocols as indicators of the discovered protocol, and protocol with higher precision and recall and smaller size are desired.

However, in real-world setting P is unknown and we need to revisit definitions of precision and recall. In such a context, we use the classical machine learning approach of *k-fold cross validation* to split our datasets into learning and testing sets [5]. 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 in this setting, as it is not clear how to measure it.

7.2 Datasets

We perform experiments on synthetic and real-world datasets:

7.2.1 Synthetic dataset

We simulated the `Retailer` service (see Figure 3 for a simplified representation of the `Retailer` service protocol) to collect the log of its interactions with clients. For this dataset, although the scenario is synthetic, the log is collected using a real-world commercial logging system for Web services (HP SOA Manager, available at `managementsoftware.hp.com/products/soa/`). Table 1 shows the characteristics of the dataset.

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 dataset (consists of 5000 conversations) was imperfect by containing swapped and missing messages, and also incomplete conversations caused by clients which left conversations or crashed. We analyzed the log and identified conversations not accepted by P, and computed the noise level as reported in Table 1. Noise level in this includes both noisy and incomplete conversations.

7.2.2 Real-world dataset

The second dataset is a real world log of interactions of a multi-player on-line game Web service called *Robostrike* (`www.robostrike.com`) with its clients. This game

	Retailer	*Robostrike*
# of operations (n)	10	32
# of conversations in CL	5,000	25,804
Arity of each operation (ϕ)	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

Table 1: Characteristics of the datasets

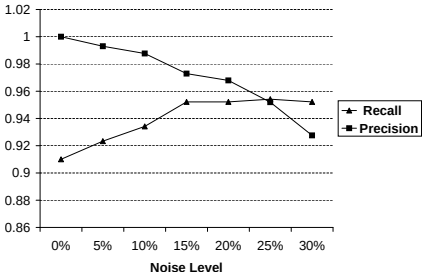
service simultaneously manages ongoing conversations with players as clients. The service has 32 operations, which clients invoke by sending synchronous or asynchronous XML messages. In this dataset, conversations range from very short (1 message) to very long (1,921 messages), depending on the time 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 dataset. We collected $25,804$ conversations over a period of two weeks. This dataset 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 25000 conversations of *Robostrike* into 5 sets of each 5000 conversations, and in each experiments used $20,000$ conversations as the learning set and the last set as the testing set to measure the quality of *DP*.

7.3 Robustness of noise identification approach

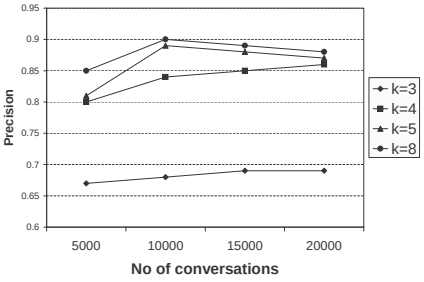
The precision and recall of the algorithm on the `Retailer` dataset is presented in Figures 10(a) for zero to 30% of noise level. In fact, the actual log contains 9.78% of noise, and we artificially introduced more random noise (of type of swapping or removing one or more messages in a connversation) to go up to 30%. 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%. However, the recall never reaches 100% as some correct but infrequent sequences are classified as noisy based on estimated θ. This is one of the reasons for having a refinement step. We used k=4 as the sequence length in these experiments. The experimental results also indicate that the approach is highly scalable in the number of conversations, depicted in Figure 10(f). The running time increases almost linearly by the conversation size.

7.4 Handling incompleteness in the service log

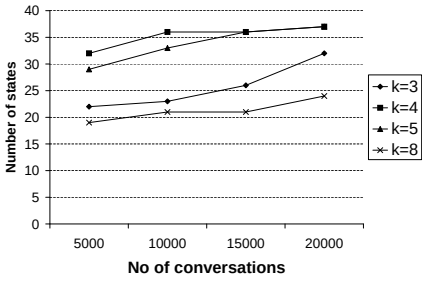
The direct application of the algorithm (steps 1-4) on `Robostrike` dataset results in the average precision of 88.7% and the size of 67.8 states ($k = 4$). Although the precision is very good, *DP* is conceived to be complex for a protocol designer to work with, due to the high number of states. Then we applied our heuristics on transparent and pervasive operations on this dataset. The result shows that out of 32 operations of the service, 9 are transparent in many states of *DP* and 3 are pervasive. The *DP* discovered after applying our heuristics has 37 states. This *DP*



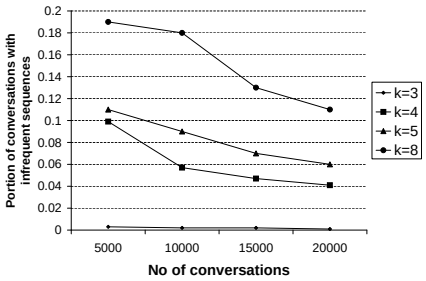
(a) Precision and Recall of *DP* for `Retailer`



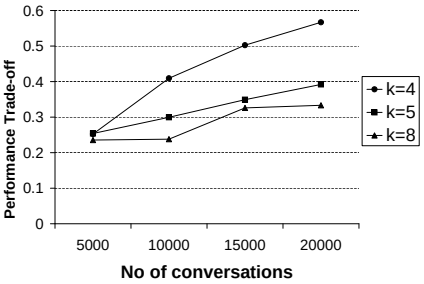
(b) Acceptance Rate of *DP* for `Robostrike`



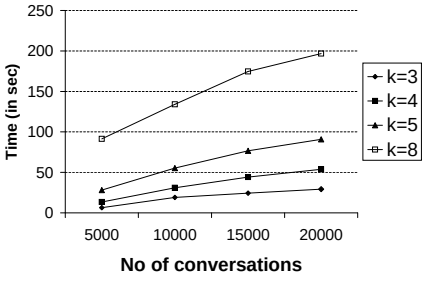
(c) Number of states of *DP* for `Robostrike`



(d) Portion of conversations in the **testing set** with at least one sequence having a support less than estimated noise threshold for `Robostrike`



(e) $k = 4$ achives the best performance trade-off in `Robostrike`



(f) Execution time of the algorithm on `Robostrike` dataset

Figure 10: Results of experiments

also achieves average precision of 86.3, however it is simpler, with less number of states.

7.5 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` dataset with different conversations size ranging from 5000 to $20,000$. In all of these experiments, we also applied our approach for handling incompleteness. Figure 10(b) shows the precision values of *DP* in these experiments. Figure 10(c) illustrates the number of states for *DP*s. Figure 10(d) depicts the portion of conversations in the testing dataset 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 a smaller portion of conversations are filtered based on noise estimation (denoted by n_k). Putting all together, we need to maximize $Precision_{DP}/(Size_{DP} * n_k)$. Figure 10(e) 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, as for $k = 3$ 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 can not 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. So, we end up with a small model that does not have a good recall but achieves a high precision. In conclusion, we recommend a value of $k = 4$ for protocol discovery in our approach, but if any particular factor, e.g., the size or precision is more important for a discovery task, then the user can use this behavior of the algorithm as a guideline to set the appropriate k value.

7.6 Refinement of the Discovered Protocol

As discussed before, it is unlikely that an automated approach discovers a model with hundred percent precision and recall. Achieving this goal requires help from the user. In this section, we explain our experiments on improving the precision and recall of discovered protocols via our refinement techniques, i.e. (i) metadata driven refinement, and (ii) classification of differences between discovered protocols and uncertain conversations (see Section 6).

7.6.1 Meta-Data Driven Refinement

The discovered protocol *DP* for `Retailer` includes a transition called `ShippingInfo` from state `Start` to state `ShippingInfo` (see Figure 9). However, this transition should not be included in *DP* as it is not a part of reference model *P* in Figure 3. It 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 Figure 5(a)). This value is above the estimated noise threshold, i.e. 0.018, for sequences of length 4 in this dataset. Computing the metadata for this transition in *DP*, it 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 our tool.

7.6.2 Distance-based Protocol Refinement

The result of analyzing uncertain conversations for `Retailer` dataset 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 transition `SubmitPO` from state `Start` to state `SubmitPO` in the protocol. This is a desired suggestion (see Figure 3), however, it has been excluded from *DP* during discovery due to low support of relevant sequences. By inspecting the first 5 classes with the highest cardinalities in the hierarchy, 83% (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` dataset ($k = 4$) there were 1947 uncertain conversations. The class hierarchy for this conversations consists of 55 classes in 7 levels. The first level (leaves) has 29 classes. The 5 highest cardinality classes contain 262, 237, 152, 130, and 126 conversations. Our refinement approach reduces the task of inspecting 1947 conversations manually to inspecting high cardinality classes at 7 levels, which is small relative to the number of uncertain conversations.

8 Related Work

The general problem of inferring a model from data samples has been studied in different contexts including grammar inference (e.g., [15, 33]), frequent pattern mining from database (e.g., [18]), schema discovery (e.g., [9, 19]) and process discovery from the software/workflow logs (e.g., [3, 6, 13, 25, 38, 42]). We study related work with respect to handling imperfection in model discovery considering the three stages of protocol discovery: noise identification and handling, model discovery, and model refinement.

8.1 Noise Identification and Handling

In the area of model discovery, the existence of noise in data is recognized in process discovery research [3,12,29,36,38]. Most of process discovery approaches use a frequency threshold to filter noise in a pre-processing step [3,12,36,38]. This is performed by manually setting a frequency threshold. In particular, the approach of Cook et. al. [12] assumes manual provision of a cut-off threshold. Agrawal et. al. [3] compute the noise threshold based on assumption that error rate of the logging infrastructure is given. Wil van der Aalst et al. in [38] assume a noise factor with the default value of 0.05. In [36] the need for automatically learning it in process logs is acknowledged.

In this paper, we have presented an approach to automatically estimate the noise threshold from the input dataset. This is significant due to the following reasons: (i) user generally has no idea of how to manually set a threshold for a given dataset, (ii) the value of the threshold is not the same for all datasets, and for all sequence lengths. For instance, the noise threshold for sequences of length 3 is different (bigger) than sequences of length 4, as frequency distribution of sequences varies with k. This value also depends on the number of conversations in the input dataset. Finally, we have experienced that setting a same fixed threshold value for various datasets, sequence length and algorithm parameters results in discovering models which either include noisy, or exclude many correct conversations.

The approach presented in Laura Maruster et. al. [29] uses machine learning based approach to learn a set of robust rules from potentially noisy dataset. 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 labelled activity pairs and corresponding frequency statistics from a noisy log. These rules are then used to predict the relationship of activities in an unseen dataset. The following characteristics of this approach makes it less suitable for applying on service logs compared to our automated approach for estimating noise threshold:

- The discovered rules specify thresholds for the values of some frequency-based parameters learned from the training dataset. To be effective, rules have to be learned from a learning set very similar to the testing set (e.g., has the same distribution and generated from the same logging infrastructure) [29]. However, our approach directly analyzes the frequency distribution of the input dataset to discover the noise threshold.

- 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.

8.2 Model Discovery Techniques

The problem of protocol discovery is similar to that of inferring grammar inference form language samples [15,33]. We focus on regular grammars as their language is represented using deterministic finite state machine machines. There are two main classes of approaches for inferring regular grammars: generalization-based, and specialization-based. In the generalization-based approach the algorithm builds the most precise representation of the target FSM (e.g., prefix tree), which has one path for each distinct input sequence, and then generalize it using merge operations [33, 37,39]. In specialization-based approach, an over-generalized simple model is built first and then its precision enhanced through node splitting. This increases the size of the model [12, 26, 33].

Software process models are much closer in terms of size to the most generalized representation of the model than the size of the most precise representation of the target model. This has been confirmed by experiments that Herbst et. al. [26] performed by applying both generalization and specialization based approaches on the log of sequential workflow models. They conclude that the number of state-merging operations that are required to perform to get from the prefix tree to the target model is much higher compared to the number of required splitting operations from the most generalized model to the target model. In addition, generalization-based approaches are best in discovering loop-free (acyclic) automata models [39]. However, it is important for the algorithm of protocol discovery to be able to discover models with loops. The initial model in specialization-based approaches by construction covers all loops that are found in conversations. For above reason, we have decided to take a specialization-based approach in the protocol discovery. One may legitimately argue that this approach does not allow for incremental learning from additional new samples. It is possible to employ a generalization-based approach to enable incremental learning, after discovering a model from instances in logs using a specialization-based approach.

Cook et. al. [11, 12] also propose a specialization-based approach for discovering sequential software process models represented as non-deterministic FSMs. We are inspired by this approach for building the initial discovered model. However, for specialization, Cook et al. approach recommends splitting only the node next to the last in an incorrect sequence. This is not applicable to deterministic FSM, discovered in the context of protocol discovery, as using it leads to non-deterministic FSM, in which the incorrect sequence is not removed, as discussed in Section 5. We proposed a splitting algorithm that recommends splitting all middle nodes of an incorrect sequence and resolves this issue.

Herbst et. al. [26] also present a specialization-based approach for discovering sequential model of process logs represented using HMM (Hidden Markov Model). In this approach all states with more than one incoming are considered as candidates for splitting. Candidate states are examined for splitting into two states with all possible disjoint set of inputs to observe if any of such models improves the log-likelihood of the model 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. However, we use an algorithmic approach which splits nodes in sequences with a support lower than the noise threshold. The complexity of our splitting method is polynomial in the order of number of sequences of length k generated from the graph G (discussed in Section 5). On the other hand, determining a right value for the threshold of log-likelihood is the job of the user.

In software engineering, researchers analyze the traces of software executions to discover execution models of software that is mainly used for debugging [6,42]. Authors in [6] use existing probabilistic automata learners to discover data and control flow specification of APIs or ADTs inside a program.

Discovering XML schemas and DTDs from sample XML files (e.g., [9,10,19]) is also related to the problem of model disocvery. The focus of this problem is rather different than protocol discovery. DTD discovery approaches are infer a set of regular expressions from XMLs that represent XML DTDs [9, 19]. Schema discovery is usually mapped to different classes of grammars, e.g., context-free grammars [10] or special subclasses of regular automaton [9].

Another related area is discovering concurrent process models form workflow logs [3, 11, 13, 22, 23, 25, 36, 38]. ProM [35] is a workflow discovery prototype tool based on some of above approaches [38]. Although these approaches allow for discovering process models with complex constructs, such as parallelism and synchronization points, they are not appropriate for protocol discovery as generally they make the assumption that each activity appears only once in the target model [38]. However, it is quite common for the same activity to appear in different part of the model, e.g., operation `Ship` in `Retailer` example. The only work in this area that relaxes this assumption is the work of Herbst et. al. [25], which presents a specialization based approach. The splitting operations introduced in this approach extends the previous work of authors for discovering sequential process models [26]. Note that our approach also takes a specialization approach. In addition, to the best of our knowledge, no approach 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 generalization of the discovered models and so to compensate for log incompletes. The work of Dustdar et. al. [16] uses existing process discovery techniques [38] to discover the composition logic of a Web service.

It should be noted that software model discovery has been also investigated through static analysis of the code (e.g., [17,41]). This approach is complementary to ours. However, the discovered model in this approach does not enjoy from the benefits of dynamic analysis in capturing the real model of service interactions, but the model prescribed by implementation.

8.3 Refinement of Discovered Models

It is unlikely for any automated approach to discover the model that is precise, and simple enough as desired by human. Another contribution of this paper is to

provide an interactive approaches (meta-data driven model pruning, and distance-based model extension) to refine the discovered protocol models to cater for inaccuracies that may have been introduced during the pre-processing or the protocol discovery visa user interaction as presented in Section 7.6. Considering the related work, our contribution is original in the sense that, 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 in [12]. In the area of software engineering, the problem of debugging software specifications is studied in [7]. In this approach, after discovering software specifications from software execution traces [6], 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.

9 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 make a number of unique contributions. We provided a characterization the problem of protocol discovery in real-world setting and captured different conceptual components of a protocol discovery solution in a framework. Given the fact that conversation logs are imperfect (contains incorrect and incomplete data), we proposed approaches to address this issue. In particular, we presented a quantitative approach and an algorithm for estimating a noise threshold, which is used to filter noisy conversations from the log. We presented a protocol discovery algorithm that also handle log incompleteness. We proposed an interactive protocol refinement technique that provides support for analysis of uncertain conversations and correcting discovered protocols. We developed a prototype system that implements the protocol discovery and refinement techniques proposed in this paper. This prototype constitutes the initial testbed model discovery solution for SOA Manager, the Web services monitoring tool provided by HP. We validated the proposed techniques using both synthetic and real service logs.

We observe that always mapping between operations in the log and transitions in the model is not 1-to-1. We are currently extending the proposed discovery techniques to derive transition conditions from logs (i.e., capture message body information and detect data conditions in model transitions) to identify the correct mapping of operations to transition in the log, i.e., the same message with different content may lead to different transitions in the model. We also plan to extend the approach to cases where the requester-provider interaction spans across multiple

services as this would be quite useful for companies to understand their interaction patterns, going beyond the point-to-point interaction between two specific services.

References

[1] Hp soa manager. www.managementsoftware.hp.com/products/soa.

[2] Web service interoperability organization. www.ws-i.org.

[3] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In *EDBT*, 1998.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Application*. Springer-Verlag, 2004.

[5] E. Alpaydin. *Introduction to Machine Learning*. MIT Press, 2004.

[6] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1), 2002.

[7] G. Ammons, D. Mandelin, R. Bodik, and J. Larus. Debugging temporal specifications with concept analysis. In *PLDI*, 2003.

[8] B. Benatallah, F. Casati, F. Toumani, J. Ponge, and H. R. Motahari-Nezhad. Service mosaic: A model-driven framework for web services life-cycle management. *IEEE Internet Computing*, 10(4), 2006.

[9] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from xml data. In *VLDB*, 2006.

[10] B. Chidlovskii. Schema extraction from xml collections. In *JCDL '02: Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, 2002.

[11] J. Cook, Z. Du, C. Liu, and A. Wolf. Discovering models of behavior for concurrent workflows. *Computers in Industry*, 53(3), 2004.

[12] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3), 1998.

[13] J. E. Cook and A. L. Wolf. Event-based detection of concurrency. In *SIGSOFT Foundations of Software Engineering (FSE)*, 1998.

[14] A. Datta and et al. Automating the discovery of as-is business process models: Probabilistic and algorithmic approaches. *Information System Research*, 9(3), 1998.

[15] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9), 2005.

[16] S. Dustdar and R. Gombotz. Discovering web service workflows using web services interaction mining. *International Journal of Business Process Integration and Management (IJBPIM)*, Forthcoming.

[17] A. Faras and Y. Gueheneuc. On the coherence of component protocols. *Notes in Theoretical Computer Science*, 82(5), 2003.

[18] M. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *VLDB*, 1999.

[19] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. Dtd inference from xml documents: The xtract approach. *IEEE Data Eng. Bull.*, 26(3), 2003.

[20] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5), 1967.

[21] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3), 1978.

[22] G. Greco, A. Guzzo, and G. Manco. Mining and reasoning on workflows. *IEEE TKDE*, 17, 2005.

[23] G. Greco, A. Guzzo, and L. Pontieri. Discovering expressive process models by clustering log traces. *IEEE TKDE*, 18, 2006.

[24] J. Han and M. Kamber. *Data mining: concepts and techniques*. 2000.

[25] J. Herbst. Dealing with concurrency in workflow induction. In *European Concurrent Engineering Conference*, 2000.

[26] J. Herbst and D. Karagiannis. Integrating machine learning and workflow management to support acquisition and adaptation of workflow models. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 9(2), 2000.

[27] J. E. Hopcroft and J. D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. 1990.

[28] D. E. Knuth. *The art of computer programming, volume 2: seminumerical algorithms*. Addison-Wesley, 1997.

[29] 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 Min. Knowl. Discov.*, 13(1), 2006.

[30] H. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, and F. Casati. Protocol discovery from web service interaction logs. In *ICDE 07*, April 2007.

[31] H. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, F. Casati, F. Toumani, and J. Ponge. Servicemosaic: Interactive analysis and manipulations of service conversations. In *ICDE 07*, April 2007.

[32] H. R. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, F. Casati, and P. Andritsos. Message correlation for conversation reconstruction in service interaction logs. Technical Report UNSW-CSE-0709, The University of New South Wales, Australia, 2007.

[33] R. Parekh and V. Honavar. Grammar inference, automata induction, and language acquisition. In *A Handbook of Natural Language Processing*, chapter 29. 2000.

[34] W. Pauw and et. al. Web services navigator: Visualizing the execution of web services. *IBM System Journal*, 44(4), 2005.

[35] Process-Mining-Group. ProM workflow mining prototype. In *http://is.tm.tue.nl/research/processmining/tools.htm*, 2006.

[36] R. Silva, J. Zhang, and J. Shanahan. Probabilistic workflow mining. In *KDD*, 2005.

[37] F. Thollard, P. Dupont, and C. Higuera. Probabilistic DFA inference using Kullback-Leibler divergence and minimality. In *ICML*, 2000.

[38] W. van der Aalst and et. al. Workflow mining: a survey of issues and approaches. *DKE J.*, 47(2), 2003.

[39] E. Vidal, F. Thollard, C. Higuera, F. Casacuberta, and R. Carrasco. Probabilistic finite state machines - part II. *IEEE TPAMI*, 27(7), 2005.

[40] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. of ACM*, 21(1), 1974.

[41] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4), 2002.

[42] J. Yang and et. al. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.