Protocol Discovery from Imperfect Service Interaction Data

Hamid Motahari[1,2], Boualem Benatallah[1], Régis Saint-Paul[1]

University of New South Wales[1]
Sydney, NSW 2052, Australia
{hamidm, regiss, boualem}@cse.unsw.edu.au

National ICT of Australia (NICTA)[2]
Sydney, NSW 1430, Australia

Abstract

This paper studies the problem of discovering protocol definitions from read-world service interaction data, which often are imperfect in various ways. It first describes the challenges in protocol discovery in such a context and the different aspects that must be considered by a protocol discovery solution. Next, it reports the current progress by presenting a discovery algorithm that is robust to log imperfection and widely applicable. Following, it shows our interactive protocol refinement approach that is intended to correct possible imprecisions introduced in the discovered protocol due to log imperfection. Finally, the experimental results on both real and synthetic data is presented.

1 Introduction and motivations

In Web service, a *conversation* is a sequence of message exchanges between two or more services to achieve a certain goal, e.g., to purchase and pay for goods. A *business protocol* of a service is a specification of the possible conversations that a service can have with its partners [2]. Endowing Web services with a definition of the business protocol allows development tools and runtime middleware to deliver functionality that considerably simplifies the service development lifecycle ranging from automated code generation and service compatibility and compliance checking to automated exception handling.

Protocol discovery refers to the process of inferring the business protocol definition from service interaction data in a service execution log. There are several scenarios in which protocol discovery is useful and needed: (i) *protocol definition:* In some cases, the protocol definition is not provided. This can happen for many reasons: e.g., the service has been developed using a bottom-up approach, by wrapping an existing application as a service for which the protocol specification is not available; (ii) *protocol verification, compliance and exception handling*: even when the designed protocol specifications are available, protocol discovery allows to verify if the service implementation follows the designed protocol, if not, what are the differences, what parts of the protocol are more often used, and what are the parts that the most exceptions occur, etc. In addition, it enables checking the compliance of service interactions with specifications required by some domain-specific standardization body or industry consortium.

Proceedings of the VLDB2006 Ph.D. Workshop, Seoul, Korea, 2006

2 Protocol Discovery Problem: Facets, Challenges and Related Work

The discovery problem has many facets. These facets include the scope of the protocol discovery, the formalism that we choose for discovered protocols, and other facets correspond to the steps of protocol discovery that are shown in Figure 1. To illustrate the concepts we use a supply chain scenario with `Retailer`, `Warehous`, `Payment`, and `Shipper` services, and clients of `Retailer` service.

2.1 Scope of Protocol Discovery

Protocol discovery can be performed from the perspective of a service provider, a service requester, or as a middleware service. Considering these different possibilities, we can discover one of the following models.

Bi-party protocols. The protocol model of a given service in interactions with another specific (type of) service. For example, the protocol of the `Retailer` service with its clients.

Service-specific view of choreography. The protocol of a given service when interacting with many or all of its partners. For example, the protocol of the `Retailer` service interacting with `Warehouse`, `Payment`, and `Shipper` services in the supply chain scenario.

Choreography. This includes all interactions among multiple services. For example, the choreography of all partner services in the above supply chain including clients. Note that a choreography model capture all interactions between the partners, e.g., interaction between the `Payment` and `Shipper` services that is not captured in the `Retailer`-specific view of the choreography.

2.2 Discovered Protocol Model

Another facet is developing or choosing a language (formal and visual) to represent discovered protocols. We adopted

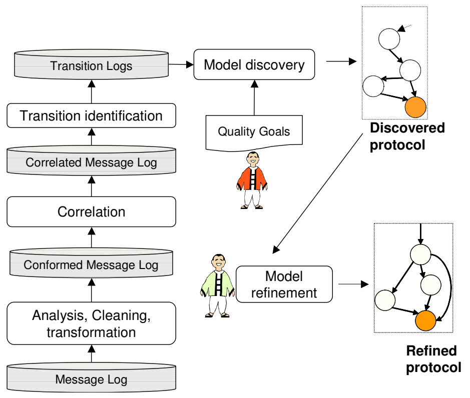

Figure 1: Protocol Discovery Process

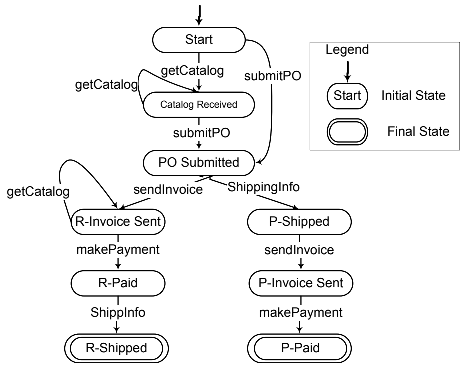

Figure 2: The business protocol of the `Retailer` service

deterministic Finite State Machines (FSMs) following our earlier work on business process modeling for Web services [4]. In a FSM-based protocol model, states represent the different phases through which a service may go during its interactions with partners. A protocol has one initial state and a set of final states. Transitions are labeled with message names with the semantics that the exchange of the message causes the transition from a source to a target state. A complete conversation is as a path from the initial state to a final state.

Example 1. Assume that the `Retailer` service 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 it 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 2. `R-Shipped`, and `P-Paid` are final states (for regular and premium customers, respectively).

As discussed in the following, discovering a business protocol from potentially imperfect conversation data is always an imprecise process. To provide the user with better understanding of the discovered protocol (*DP*), we extend the FSM model by attaching metadata in form of support and confidence to states and transitions. The support of a state expresses the probability that a conversation in the log terminates in that state. The confidence of a state gives the probability of termination of those conversations that traverse it in that state. The support of a transition is defined as the probability that conversations in the log take that transition. The confidence of a transition is defined as the probability that the transition is taken among all other possible transitions from its source state.

2.3 Message Log Characteristic and Issues

Dustdar et. al. [7] have described the different ways that service interaction log can be collected. Depending on how services are implemented and on the type of tools used to monitor service executions, different kind of information may be included in the logs, which can make protocol discovery more or less complex. In practical scenarios, there are three different kinds of information that can be included in a conversation log: (i) message transfer information (sender, receiver, timestamp), (ii) SOAP message header; (iii) SOAP body (message content); We formally define a message log format as the following: a message log *ML* is a set of entries $e = (s, r, \tau, m(mh, mb))$, where s and r denote the sender and the receiver of message m, and τ is the timestamp. mh and mb stands for header and body of message m, which may or may not be present.

In practice, there maybe other information in the log that we do not discuss them, here, for brevity. We refer to the above message log format as the common format. Given this format, the first step of the protocol discovery process consists of collecting data from different data sources (shown as "Message Log" in Figure 1) and transforming it into this format (called "Conformed Message Log" in Figure 1). There are some challenges in data transformation, though interesting, fall out of the scope of this work.

The next pre-processing step is the construction of conversation log *CL* from *ML* (referred to as "Correlated Message Log" in Figure 1). This step involves identifying and correlating messages belong to the same conversation. This require existence of a conversation identifier, denoted *cid*, for each entry in *ML*. Assuming such identifier exists or could be inferred, which we discuss later in this section, we can transform *ML* to conversation log *CL*. Formally, a conversation log $CL = \{c_i | 0 \leq i \leq k\}$, is a set of conversations in which conversations c_i is a sequence $< e_1, e_2, ..., e_n >$ of *ML* such that, for any u, v, $0 \leq u, v \leq n$: (i) $e_v.cid = e_u.cid$ (same conversation identifier), (ii) $e_u.\tau \leq e_{u+1}.\tau$ (entry timestamps define the sequence order). However, in practice there are a number of challenges that make the realization of the above 2 steps hard, and generally the protocol discovery, a challenging problem:

Log imperfection. The cleaning and transformation step may introduce some imperfection in form of missing or incorrect information about messages in *ML*. In addition, conversations of *CL* may contain different types of imperfection: disordered messages, incomplete conversations (conversations that terminate in a non-final state), missing messages, etc. These imperfections can happen for various reasons including bugs and flaws in the implementation of the logging infrastructure or the service, interruptions in logger, exceptions in the service execution, network failure, abnormal termination of the service interactions, and finally incorrectness or imprecision of the timing information [11].

Correlation. Correlation refers to the process of identifying which message in the log belongs to which conversation between services. Currently, there is no widely accepted approach for incorporation of correlation information in the log of Web services. Disparate service management infrastructures deal with this problem differently ranging from customized ways of using standard proposals (e.g. WS-Addressing), introducing propetriary approaches (e.g., IBM WebSphere deploys a nested transaction identifier mechanism that can track the interactions inside the same organization), or providing no support and leaving it to be addressed at the service implementation level by service developers. When such an identifier do not exist, in our vision, we can map the correlation problem to that of record linkage (or entity resolution) in the database community [5].

Transition identification. Transitions between states of a protocol can depend not only on messages, but also on message parameters. Furthermore, a protocol model may also allow timed-controlled transitions or other sophisticated modeling constructs. This means that the mapping between messages in the log and state transitions in a conversation may not be 1-1. Hence, part of the protocol discovery process lies in identifying state transitions. The output of this step is called *transition log*. We believe this problem can be mapped to that of associate rule mining in the database community [9], in the sense that the conditions on transition can be discovered as a set of association rules attached to that transition. This step could be done as a post protocol discovery step in order to refine the discovered protocol model. In the existing work, the work of decision mining for workflows in [12] uses classification approaches and in particular decision trees to discover simple rules for an already known workflow model from events in the log, however, the idea there is to tag the workflow transitions with conditions discovered from the log, but not to refine the discovered protocol model.

Performance. Another issue is the effect of logging on the performance of running services and also the fact that performance issue may lead to imperfection in logs. This is mainly an issue if we log the full payload of messages, as well. Our experiment shows that logging message payload increases the chance of having imperfection for messages that are close in time, e.g., such messages may get the same timestamp and so it is difficult to judge on their ordering.

2.4 Algorithmic Aspect of Protocol Discovery

The general problem of inferring a model from instances is not new. It has been extensively studied in the context of grammar inference [10], software process [6, 1] and workflow mining [14]. However, very few existing approaches tackle the problem of model discovery from imperfect logs as characterized above.

If we consider conversations as the strings of a regular language, we can map protocol discovery to that of grammar (autoamta) inference [10]. The main two approaches in this area are based on state merging and state splitting. State-splitting approaches start from a generalized representation of the target automata and iteratively specialize the model to maximize a fitness measure. In contrast, state-merging approaches start from a specialized representation of the model (most often a prefix tree) and iteratively generalize it. Unfortunately, according to Gold's Theorem [10] this problem is *undecidable* in limit even if the sets of positive (accepted) and negative (rejected) samples of the target language are given. On the other hand, in most real-world applications, like in protocol discovery, only positive samples are available. In this case, approximate probabilistic grammar inference approaches becomes tractable[10]. However, these approaches are ineffective for model discovery from imperfect log. In this area, [13] extends grammar inference approaches for inferring a probabilistic automata from noisy language instances, however, this approach cannot detect noise or remove it, but applies more restrictive state merging rules that leads to less merges and so a more complex model.

Workflow mining from event logs has also been subject of research for many years [14]. However, the focus of the research in this area is rather different. Some workflow discovery approaches assume the existence of a complete log and provide algorithms to discover complex process constructs such as complex loops, paralelism and non-free choice.

In the area of the software process mining, [6] proposes a state-splitting based approach for discovering software processes from software execution streams. They recognize noise in logs and ask the user to specify a noise threshold that is used to filter noise. In the area of Web services, the work of Dustdar et. al.[7] considers the problem of service interaction mining. In fact, they map message interactions to the input format accepted by workflow mining tools and uses such tools to discover the service workflow models. Finally, model discovery by static analysis of code [8] is also relevant. While this approach is applicable to well-structured (e.g., block-structured) code, it does not enjoy from the benefits of dynamic interactions analysis.

2.5 Model Refinement

The fact that logs' data is imperfect makes it impossible for the algorithm to discover the exact protocol model supported by a service. 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 and also the analysis of conversations in the logs that cannot be accepted by *DP*. The challenge here consists in how to best guide users through the areas of uncertainty and how to analyze excluded conversations in a way that is as simple and as effective as possible.

3 Research Progress and Results

We argue that it is important for any approach to protocol discovery to characterize its position with respect to the various facets presented in Section 2. In terms of *scope*, up to now we have tackled discovering bi-party protocol models. The *modeling formalism* is as described in Section 2.2. In terms of *correlation*, at this stage we assume that conversation identifiers are inserted either by service developers or by the logging system. In terms of *assumptions on conversation logs*, we consider most “real life” scenarios, whenever Web service interactions are monitored via Web services monitoring engine such HP SOA Manager or IBM WebSphere Process Server. Until now, we have made three unique contributions: (i) providing an automatic noise estimation method to deal with log imperfection, (ii) developing an algorithm for protocol discovery, and (iii) providing a methodology for refinement of the discovered protocol. In the following, we give a short overview of our approach and omit the technical details due to lack of space.

3.1 Measuring Noise in Message logs

We assume that occurrence of imperfect conversations follows a *Poisson distribution* (infrequent and non-repeatable events). The challenge lies in defining the meaning of “infrequent” and a mechanism for identifying a frequency threshold, which is widely applicable. Instead of conversations we consider subsequences of length k (called k-sequence) of conversations for two reasons: (i) computational complexity, (ii) typically few subsequences of an infrequent conversation are infrequent but other correct subsequences of it can contribute to better discover the protocol model. For each k-sequence, we compute its support as the percentage of conversations that contain that sequence. Then we analyze the support distribution of k-sequences. The intuition confirmed by experiments is that a lot of incorrect sequences share a similar low support. On the other hand, correct sequences are fewer in number and have greater (typically dissimilar) support. Ranking the the sequence distribution by support identifies a step function. From the low end of this function, we take the first support that has a smaller length (sequences with the same support value) relatively to the increase in the value of the support from the previous support as a threshold. This approach proved robust in many experiments.

3.2 Protocol Discovery Algorithms

Our protocol discovery algorithm is similar to that of software process mining [6] algorithm in the sense that it is a state splitting based algorithm that starts from a generalized model of the protocol and iteratively refines it by splitting over-generalized states, i.e., states that allow generation of sequences that are not in the input log data. The goal of splitting is to disallow such sequences (starting splitting from sequences of length 2 to k in the generalized protocol model). Though, the main differences of our algorithm with [6] is that we do not require the user to provide a noise threshold as the input, as we estimate it adaptively for sequences of different lengths during the splitting. The other difference is that we devised a new splitting procedure, as the splitting method in [6] can not be used to disallow incorrect sequences in a deterministic state machines and it causes non-determinism. The technical details of the algorithm is omitted for brevity. One main property of the algorithm is that its time complexity is polynomial.

There are two main criteria when evaluating the quality of a discovered protocol: its simplicity and its fitness [3]. The former, with FSM-based models, can be defined as the size (measured as the number of states) of the discovered protocol. For the later, since a protocol can be seen as a classifier, it can be assessed by the classical *recall* and *precision* measures. The maximal recall ($rec(DP) = 1$) is achieved when all the correct conversations (CC) present in the conversation log are accepted (AC) by DP. Similarly, a protocol DP has a maximal precision ($pre(P) = 1$) when all AC are correct ones. More formally:

$$rec(DP) = \frac{|AC \cap CC|}{|CC|}, pre(DP) = \frac{|AC \cap CC|}{|AC|}, Q_e(DP) = \frac{rec(DP) \times pre(DP)}{size(DP)/size(P)}$$

In this formula, we consider that P is the minimal protocol model. The quality measure, denoted by Q_e, can only be used in an experimental setting. Indeed, in real situation, the reference protocol model P is unknown. We used this measure to assess the robustness of the model and also to evaluate the effectiveness of noise estimation approach.

3.3 Protocol Refinement

Threshold cannot always provide a clear cut separation between correct and incorrect conversations. Consequently, the discovered protocol may allow generation of some incorrect conversations or exclude generation of other infrequent correct conversations. To overcome these limitations, we devise some methods for interactive refinement of discovered protocols by featuring (1) *meta-data driven protocol refinement* and (2) *classification of uncertain conversation for interactive protocol refinement*.

(1) ***Meta-Data Driven Protocol Refinement.*** The goal of this step is to improve the precision of DP. By browsing the discovered protocol and visually examining associated meta-data, users can understand potential errors made during the discovery process and correct them. We use the metadata to help the user in visually navigating to the areas that needs more attention. Transition and state are respectively highlighted using various arrow thickness and state color brightness based on protocol metadata (see Section 2.2).

(2) ***Classification of Uncertain Conversation.*** The goal of this step to increase the recall of DP by inclusion of conversations that are incorrectly excluded. In our approach, the analysis of excluded conversations is based on measuring the *edit distance* between them and the protocol. In a

	Retailer − 1	*Retailer* − 2	*Game*
# Operations	8	10	32
# Conversations	5,000	5,000	25,791
Noise Level	7.64	9.78	NA
Min. Length	2	1	1
Avg. Length	5.2	9.45	43.31
Max. Length	18	50	1,921

Table 1: Characteristics of the datasets

nutshell, the distance is computed by counting the number of edit operations that have to be performed on the conversation string to obtain a conversation that is accepted by the protocol. Based on such an analysis, we group the conversation with the same set of edit operations. Each class proposes some edit operations (of type of state and transition addition) to the protocol. The user can accept or reject the changes in a visual environment to refine the protocol.

3.4 Implementation and Result

Since our approach focuses on the discovery of protocol from imperfect conversation logs, we examined its robustness with respect to the noise level present in various datasets. We used two datasets in our experiments, two synthetic and one from a real-world service. For the synthetic datasets, we simulated two real-world retailer services similar to, but more complex than, the the `Retailer` service in Example 1. We used HP SOA Manager as the logging infrastructure. We developed the `Retailer-1` and `Retailer-2` services and their clients in Java using Apache Axis as the SOAP engine, and Apache Tomcat as the Web server. The protocol discovery framework has been implemented in Java as Eclipse plug-ins. The real-world dataset is taken from the user's interactions log of a game service (called *robostrike* (http://www.robostrike.com)). This dataset allows us to evaluate the capabilities of our prototype to handle large datasets (several gigabytes) as well as discovering an unknown and complex protocols (see Table 1).

For the synthetic scenarios, we also introduced some artificial noise to reach the noise level to 30% from 7.64 abd 9.78 in the datasets. Our experiment show that the noise level does not sensibly affect the precision and recall of our algorithm. In both cases the precision decreases from arround 97 to 92 percent, and recall increases from arround 88 to 95 percent for zero to 30% of noise, respevtively (we ommitted the graph details for the sack of space). We also compared the size of the *DP*s of `Retailer-1` and `Retailer-2` relatively to the size of their corresponding reference models. We observed that the noise affects the size of the discovered protocols to the point that their size can be less than doubled when the noise level in the log reaches 30%. For the game service, we collected $25,791$ conversations. The model discovered by the algorithm has 68 states and 181 transitions. This result was very promising in the sense that we asked the data provider to compare the discovered model with the real protocol of the game service. The algorithm was able to identify the main paths of the real protocol of this service. The large number of states, in this case, is due to the fact that some messages can be sent in any state of the game service. For example, players can at any time send discussion messages to their partners. In addition, this service has quite large number of operations (32 operations) and the number of paths supported by the protocol is also high.

4 Concluding Remarks

In this paper, we presented the problem of protocol discovery from the real-world service interaction logs that can be imperfect in many ways. We identified challenges related to capturing, pre-processing and analyzing message logs. We identified that two of these challenges, namely message correlation and transition identification could be linked to well-known problems in the database area, i.e., record-linkage and association rule mining, respectively. We also presented our current result, i.e., a protocol discovery algorithm, which is robust to log imperfection, and an interactive protocol refinement method for refining discovered protocol models. As the future work, we intend to investigate the challenges identified in the paper, and also extending the scope of our work by discovering service-specific view of choreography models and choreographies of services from service interaction data.

References

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

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

[3] D. Angluin and C. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys (CSUR)*, 15(3):237–269, 1983.

[4] B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data and Knowledge Engineering (DKE)*, 2005, 2005.

[5] O. Benjelloun, H. Garcia-Molina, Q. Su, and J. Widom. Swoosh: A generic approach to entity resolution. Technical report, Database Group, Stanford University, 2005.

[6] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3):215–249, 1998.

[7] S. Dustdar, R. Gombotz, and K. Baina. Services interaction mining. Technical Report TUV-1841-2004-16, Technical University of Vienna, 2004.

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

[9] M. K. J. Han. *Data Mining: Concepts and Techniques*. Elsevier Science, 2006.

[10] R. Parekh and V. Honavar. Grammar inference, automata induction, and language acquisition. In *A Handbook of Natural Language Processing: Techniques and Applications for the Processing of Language as Text.*, chapter 29. Marcel Dekker, USA, 2000.

[11] W. Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. Morar. Web services navigator: Visualizing the execution of web services, 2005.

[12] A. Rozinat and W. van der Aalst. Decision mining in business processes. Technical Report BPM-06-10, BPM Center Report, 2006.

[13] M. Sebban and J. Janodet. On state merging in grammatical inference: A statistical approach for dealing with noisy data. In *ICML*, pages 688–695, 2003.

[14] W. van der Aalst, B. van Dongen, J. Herbst, L. Maruster, G.Schimm, and A. Weijters. Workflow mining: a survey of issues and approaches. *DKE Journal*, 47(2):237–267, 2003.