Protocol Discovery from Imperfect Service Interaction Logs

H.R. Motahari-Nezhad, R. Saint-Paul and B. Benatallah
University of New South Wales, Australia
{hamidm,regiss,boualem}@cse.unsw.edu.au

F. Casati
University of Trento
casati@dit.unitn.it

1. Introduction

In service-oriented computing, a *conversation* is a sequence of message exchanges between two or more services to achieve a certain goal, for example 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]. Understanding the protocol model of a service is very important, for several reasons: (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, (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 [4, 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 *interaction logs* (logs storing messages exchanged among services).

This paper deals with the problem of discovering protocol models by analyzing real-world interaction logs. There are several scenarios where protocol discovery is useful and needed: (i) In practice, the protocol definition may not be available. This can happen for many reasons, e.g., the service has been developed using a bottom-up approach, by simply SOAP-ifying an existing application. (ii) even when the protocol model is available, protocol discovery is important as we may want to verify if the designed protocol model is what is actually being supported by the implementation and, if not, what are the differences. An instance of this problem involves discovering if the service is compliant with the protocol specification required by some domain-specific standardization body or industry consortium.

Protocol discovery is also a very challenging endeavor. One problem is that interaction logs are often imperfect: they contain noise (e.g., missing messages) and are incomplete, i.e., do not contain traces of all the possible conversations. Handling imperfections is essential, as otherwise protocol discovery is useless in practice. One of the contributions of this paper consists in identifying the different kinds of log imperfections and in proposing a quantitative measure and an effective algorithm to determine noisy conversations. In a nutshell, this is done by ranking the different conversations (or subsets of conversations) in the log based on their frequency. Then we compute histograms that show the percentage increase of this frequency in going from less frequent to more frequent conversations. We show that with typical noise patterns and with typical noise introduced by commercial logging systems, noisy conversations end up in the portion of the histogram characterized by high, sparse spikes with high spread.

The next challenge is devising a discovery algorithm that has acceptable complexity and that derives, wherever possible, a model of manageable size given incomplete conversation logs in real-world scenarios. In our approach, as we adopt finite state machines (FSM) for modeling protocols, manageable size translates in small number of states. Our contribution here (Section 3) lies in proposing a novel approach that generates protocols of small sizes by leveraging algorithmic minimization techniques as well as heuristics (e.g., we identify messages whose occurrence does not change the conversation state, as these can be modeled in a very compact way in FSM-based protocol models).

Log imperfections make it in general impossible to always derive the correct protocol model. Hence, the next challenge lies in how to assist users in refining the discovered protocol. The problem is that, with logs of large sizes, it is unrealistic to ask the user to validate each possible conversation allowed by the model. Therefore, as another contribution, we propose an approach to *user driven protocol refinement* (See [7] for details). It is based on defining (i) a notion of *distance* between protocol models and conversations that cannot be generated (i.e., accepted) by the model and (ii) *manipulation operations* that can modify these conversations so that they can be accepted. We observe that the refinement process is facilitated by having users analyze manipulation operations rather than conversations, and doing so starting from operations that occur frequently and in conversations at smaller distances from the protocol.

We have validated the approach via experiments (Section 4) performed on logs of various protocols obtained from actual service execution logs. Finally, in Section 5, we discuss related work and summarize our findings. The approach has been implemented as part of a larger platform for the analysis and management of service-based interactions, called *ServiceMosaic* [4, 8], and on top of HP SOA Manager, a commercial service management platform.

1-4244-0803-2/07/$20.00 ©2007 IEEE

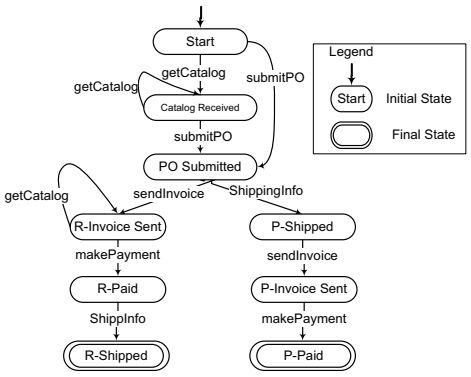


Figure 1. The business protocol of the `Retailer` service

2. Preliminaries and Assumptions

Modeling Business Protocols. We adopt deterministic Finite State Machines for modeling business processes [4]. FSM is a well-known paradigm with established formal foundations that allow for formal analysis of protocol models. However, we stress that many of the concepts presented here apply regardless of the protocol formalism.

As an example, 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 1. `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.

Web service logs characteristics. We assume that the interaction logs contain: (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 logged by commercial service monitoring tool. The only exception is that some tools do not log conversation IDs. Finding the conversation in some cases is very challenges, and we plan to investigate it as a future work [7].

Log imperfections. We call a log L *perfect* with respect to a protocol P if all conversations in L are in fact instances of P and if all possible conversations that can be generated by P (modulo loops) are present in L, if a *loop* in a conversation is defined as a substring that appears consequently two or more times, e.g., *aaa* or *abab*. In practice, conversation logs are *imperfect* for three reasons. First, they may include *noisy* conversations, that is, conversations that cannot be generated by P due to missing messages or to incorrect message timestamp information. These can happen because of (i) bugs or temporary failures in the logging system, (ii) because of imprecision of timing information. Noisy conversations are problematic as they lead to the generation of models that allow conversations that are not in fact supported by the service. As mentioned, it also causes the model complexity to increase significantly.

Second, the log may include conversations that are *incomplete*, that is, conversations that do not finish in a final state, and hence cannot be *accepted* by P. They can be present due to failure in the log system or to clients abandoning the interactions. Incomplete conversations lead to incorrect determination of which states are final in the discovered protocol. Third, a log may be *non exhaustive*, i.e., may not have all possible conversations generated by P. We also show Section 3.3 why this issue can lead to a significant increase in protocol size.

3. Discovering Protocols from Imperfect Logs

The protocol discovery approach we propose has two stages: i) analyzing the log to automatically estimate which conversations are noisy, and ii) deriving a protocol that can accept conversations estimated to be correct. The protocol derivation part also addresses the problem of non-exhaustive logs via discovery heuristics.

3.1 Estimating noise in message logs

The noise estimation process assumes that noise in the logs is random and follows a Poisson distribution. Furthermore, with respect to a noiseless log that typically has a relatively low number of distinct conversations (modulo loops), a noisy log may have many different conversations, as noise creates new message sequences in affected conversations. Intuitively, this leads to the fact that infrequent conversations are more likely results of noise. The challenge lies in defining what "infrequent" means, that is, identifying a frequency threshold. A basic solution consists in asking log administrators to provide the frequency threshold (as done in [5, 10]). This approach is not practical as users may not know the threshold, also because it depends on many factors (bugs, failures, timing inconsistencies). Hence, we propose to estimate this threshold automatically. We also observe that computing a frequency count directly on conversations is not appropriate. This is because conversations can be long, and in the presence of noise, the number of distinct conversations can be very high. In this case the frequency of each distinct conversation is low. Hence, using the frequency of each distinct conversation is not a good measure for identifying noisy conversations. Instead, we base our analysis on the frequency of message *sequences* of small length (size) k. The problem is then twofold: (i) select an appropriate value for the size k, and (ii) identify which of the sequences found in the log are correct.

1 **for** $i = 2..k-1$ **do**
  1.1 **foreach** node v of G **do**
    1.1.1 $IS \leftarrow IS \cup il(v, i+1)$
  1.2 $Prefix \leftarrow$ unique i-length prefix sequences in IS
  1.3 **foreach** $seq \in Prefix$ **do**
    1.3.1 $IE \leftarrow$ all v', $(seq, v') \in IS$
    1.3.2 **foreach** node v in $seq[2..i]$, $2 \leq j \leq i$ **do**
      1.3.2.1 $seq[j]' \leftarrow$ copy of $seq[j]$ in G
    1.3.3 **foreach** node $seq[j]$ in $seq[2..i]$ $2 \leq j < i$ **do**
      1.3.3.1 if $(j > 2)$ create an edge from $seq[j-1]'$ to $seq[j]'$ in G
      1.3.3.2 copy outgoing edges of $seq[j]$ to $seq[j]'$,
        except the edge from $seq[j]$ to $seq[j+1]$
    1.3.4 copy outgoing edges of node $seq[i]$ to node $seq[i]'$,
      except for edges to nodes in IE
    1.3.5 remove edge $(seq[1], seq[2])$
    1.3.6 add edge $(seq[1], seq[2]')$
    1.3.7 update IS
  1.4 remove all vertexes not reachable from the `Start` vertex.

Table 1. The sequence splitting procedure

For the size of the sequences, we have learned through experiments (see Section 4) that appropriate values typically range between 3 and 5 for most practical situations. Next, we turn to the problem of estimating the frequency threshold (denoted as θ) for sequences of size k. For each distinct sequence, we compute the support, denoted *supp* as ratio of the number of conversations that contain this sequence divided by the total number of conversations. It is used to order n unique sequences $Q = (q_1, q_2, \ldots, q_n)$ such that $\forall i < n, supp(q_i) \leq supp(q_{i+1})$. The intuition, confirmed by experiments, is that a lot of incorrect sequences share a very low similar support. On the other hand, correct sequences are fewer in number, have a greater support, and it is unlikely that two distinct sequences share exactly the same support unless they are part of the same longer sequence. Even in this case, the number of sequences with the same support is small.

We then proceed by generating the frequency histogram of sequences ordered by support. 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 change. 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 ϕ_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 ($\phi_m = \frac{l(m)}{n \cdot supp(q_m)}$). 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. $\phi_{m_0} < 1$)), and we set $\theta = supp(q_{m_0})$. This approach works very well in real datasets, and it is consistent with the intuition. In addition, this function is "robust" in the sense that the index m for which $\phi_m < 0.5$ or $\phi_m < 2$ is about the same (the difference concerns only a small number of sequences when compared with the total number).

3.2 Approximate Protocol Discovery Algorithm

The goal of the algorithm is to find, given an interaction log L and an estimated noise threshold, the minimal business protocol *DP* (minimal deterministic FSM) so that: (i) *DP* accepts all conversations $CC = \{c_i \in L, 0 \leq i \leq k\}$ which are correct and, (ii) *DP* does not accept any conversation c that is not in CC (modulo loops). Finding such minimal and accurate model is *undecidable* in limit [9]. So, for sequences of length k, an approximate *DP* is built from the set of correct sequences $CS_k = \{q \in Q | supp(q) \geq \theta, |q| = k\}$ in the following four steps.

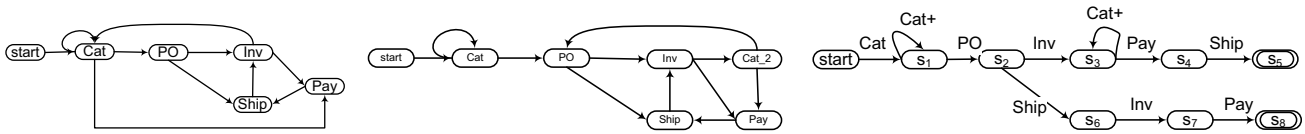
Step 1: Construction of a message graph. In this step, we build a generalized representation of *DP* as a directed graph G, in which initially one node is created for each unique message in the log. We use all sequences $q \in CS_k$ to connect these nodes in G. For example, from sequence q_1 =<`Cat,PO,Inv,Pay`> (for $k = 4$) we create a directed edge from node `Cat` to node `PO`, from `PO` to `Inv`, etc. Figure 2(a) shows the initial graph G for `Retailer` service log (Figure 1).

Step 2: Enhancing message graph precision. The initial graph G is often *overgeneralized*: it accepts, by construction, all sequences of CS_k, but also sequences which are not in CS_k. For example, the graph of Figure 2(a) allows the generation of sequence <`Start,Cat,Pay`>, which is not in CS_k. So, this graph has a low precision.

We enhance the precision of G through splitting nodes in incorrect sequences of length k generated by G using the algorithm outlined in Table 1. In this algorithm, *IS* stands for the list of Incorrect Sequences and $il(v, i+1)$ is the list of incorrect sequences of length $i+1$ from node v in G. The variable $Prefix$ keeps the set of all unique prefix sequences of length i in *IS*. 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. For example, for sequence q_1, the middle nodes include node `Cat`. We split it into two nodes (see Figure 2(b)) by creating another copy called `Cat_2` that has the same outgoing edges except for the one to `Pay`. Then, we remove the edge from `Start` to `Cat` and create an edge from `Start` to `Cat_2`. Finally G does not allows for generation of the sequence <`Start,Cat,Pay`>.

Step 3: Conversion into a FSM. We convert the directed graph to a finite state machine by naming transitions to the name of their target nodes. The resulting graph is a deterministic FSM.

Step 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 due to two reasons: (i) the graph generated in step 1 may not be the minimal form of *DP*, and (ii) splitting may cause the generation of equivalent states. For example, in Figure 2(b), nodes `Inv` and `Cat` are equivalent. They have the the same



(a) Initial graph G (b) Removing overconnections of the `Cat` node (c) Discovered protocol DP

Figure 2. Results of applying the algorithm steps on `Retailer` service log

outgoing transitions (excluding the loop), which means that they can be merged without changing the FSM properties. Hence, in this step we minimize the FSM using the Ullman-Hopcroft minimization algorithm [6]. Figure 2(c) shows the discovered protocol model of the `Retailer` service. Note there is no `submitPO` transition from the `Start` state as its support is below θ.

3.3 Discovery Heuristics

Despite the minimization, experiments (see Section 4) revealed that *DP* may be very large. We observe that there are operations in *DP* that are *transparent* with respect to a state s (their invocation, when in state S, does not cause a transition out of s). These are very common in protocols, and can be modeled as self-transitions, without requiring additional states. However, since the log is not exhaustive, 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. As a result, their modeling requires additional states.

In addition, it often happens that transparent operations are also *pervasive*, that is, they can happen in any state, i.e., are transparent in any state. 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. To handle this issue, we identify transparent and pervasive operations in the log (See [7] on details of how it is performed) and use them to generalize *DP* so to achieve a simple *DP* (with fewer number of state and transitions).

4. Experiments and Validation

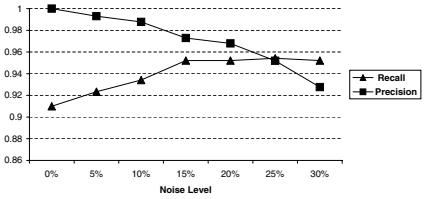
Quality of discovered protocols. There are two main criteria in evaluating the quality of a discovered protocol *DP*: one is *simplicity*, that is measured as the size (number of states) of the *DP*. The other is its accuracy. In experimental settings, where the reference model P is known, accuracy is measured through classical *recall* and *precision* metrics [3]. 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*. In real-world datasets, the reference model P is not available. So, we use the classical machine learning approach of *k-fold cross validation* to split our datasets into learning and testing sets [3]. Then, as an indication of accuracy, we define the *precision* as the average percentage of conversations in the testing sets that are accepted by *DP*.

Datasets. We simulated the `Retailer` service with 10 operations (Figure 1 shows a simplified representation of the `Retailer` protocol) to collect the log of its interactions with 10 clients. Although the scenario is synthetic, the log is collected using HP SOA Manager, available at `managementsoftware.hp.com/products/soa/`). This dataset consists of 5000 conversations has a noise level of 9.78%. It contains disordered and missing messages, and also incomplete conversations. 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. The service has 32 operations, which clients invoke by sending (a)synchronous XML messages. We collected $25,804$ conversations over a period of two weeks.

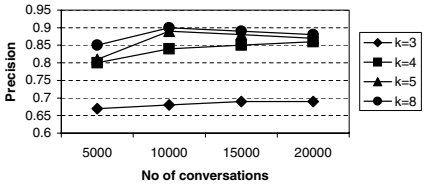
Robustness of noise estimation. Figures 3(a) shows precision and recall of the algorithm on the `Retailer` dataset for zero to 30% of noise level. As it can be seen, the log imperfection does not sensibly affect the precision and recall of *DP* as they always stay above 90%.

Performance of the algorithm on Robostrike dataset. We split 25000 conversations of *Robostrike* into 5 sets of each 5000 conversations to be used as 4 learning and one testing set. The direct application of the algorithm (steps 1-4) on this 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, 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* also achieves average precision of 86.3, however it is simpler. Figure 3(f) show that the algorithm is also scalable and the running time increases almost linearly by the conversation size.

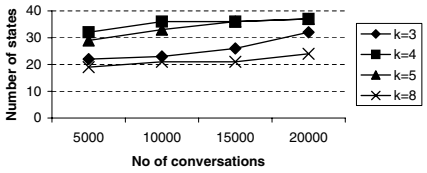
Impact of k-value on the performance. To evaluate the performance of the algorithm with different k values, we executed the algorithm for values $k = 3,4,5$, and 8 on



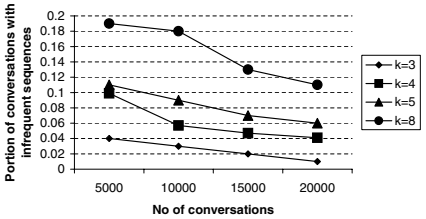
(a) Precision and Recall of *DP* for `Retailer`



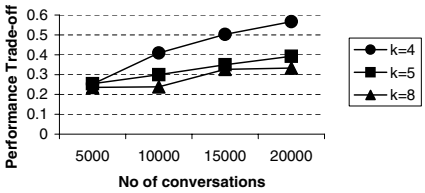
(b) Precision of *DP* for `Robostrike`



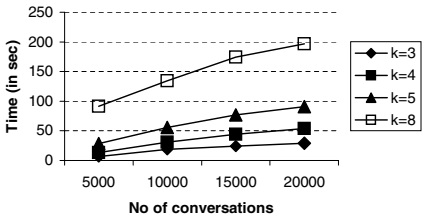
(c) Number of states of *DP* for `Robostrike`



(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 3. Results of experiments

`Robostrike` dataset with different conversations size ranging from 5000 to 20,000. In all of these experiments, we also applied our discovery heuristics. Figure 3(b) shows the precision values of *DP* in these experiments. Figure 3(c) illustrates the number of states for *DP*s. Figure 3(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 3(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. For higher k values ($k = 5, 8$), the precision gets 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 a small number of sequences. Hence, we end up with a small model that does not have a good recall but achieves a high precision. In conclusion, we recommend value $k = 4$ 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.

5. Related Work and Summary

The general problem of inferring a model from samples is not new and has been studied in different contexts including grammar inference [9], and process/workflow discovery [5, 1, 10]. The present work is inspired from the work in grammar inference [9] and process discovery [5], however, the main differences with respect to the papers above correspond to our major contributions, namely (i) characterizing and handling log imperfections, (ii) proposing a discovery algorithm robust to noise and that minimizes the model size based on heuristics derived from real-world experiments, and (iii) introducing a refinement phase, which is assisted by a framework and a tool that guides and minimizes the effort by the user through a task that would otherwise be unmanageable.

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

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

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

[6] J. E. Hopcroft and J. D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley, 1979.

[7] H. Motahari, R. Saint-Paul, B. Benatallah, and F. Casati. A framework for protocol discovery from real-world service conversation logs. *Technical Report #0623, CSE School - UNSW, Australia*, 2006.

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

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

[10] W. van der Aalst and et. al. Workflow mining: a survey of issues and approaches. *DKE Journal*, 47(2):237–267, 2003.