Adapting Web Service Interfaces and Protocols Using Adapter Simulation

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

Axel Matren, Francisco Curbera
IBM TJ Watson Research Center
New York, USA
{amarten|curbera}@us.ibm.com

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

TECHNICAL REPORT
UNSW-CSE-TR-0707

February 2007

Abstract

In today's Web, many functionality-wise similar Web services are offered through heterogeneous interfaces (operation definitions) and business protocols (ordering constraints defined on legal operation invocation sequences). The typical approach to enable interoperation in such a heterogeneous setting is through developing adapters. There have been approaches for classifying possible mismatches between service interfaces and business protocols to facilitate adapter development. However, the hard job is that of identifying, given two service specifications, the actual mismatches between their interfaces and business protocols.

In this paper we present novel techniques and a tool that provides semi-automated support for identifying and resolution of mismatches between service interfaces and protocols, and for generating adapter specification. We make the following main contributions: (i) we identify mismatches between service interfaces, which leads to finding mismatches of type of signature, merge/split, and extra/missing messages; (ii) we identify all ordering mismatches between service protocols and generate a tree, called *mismatch tree*, for mismatches that require developers' input for their resolution. In addition, we provide semi-automated support in analyzing the mismatch tree to help in resolving such mismatches. We have implemented the approach in a tool inside IBM WID (WebSphere Integration Developer). Our experiments with some real-world case studies show the viability of the proposed approach. The methods and tool are significant in that they considerably simplify the problem of adapting services so that interoperation is possible.

1 Introduction

While standardization in Web services has proved effective for integration at the lower levels of interoperability stack, interoperation at the level of service interfaces and business protocols is still a challenge due to the heterogeneity of service specifications, developed by different teams or companies. Service interfaces (often syntactically specified in WSDL [26]) declare all operations of a service. Business protocols define ordering constraints on the allowed operation invocation sequences [5, 4, 8]. In today's Web, services that are similar in terms of functionality are offered through different interfaces and protocols. The default approach for a company, using one of such services, in switching to another similar service is to develop new clients for the new service. This approach is often time consuming, costly and does not always allow for reusing previous implementations.

An alternative to developing new clients is that of developing service *adapters*. Service adaptation refers to the process of generating a service (the adapter) that mediates the interactions among two services with different interfaces and protocols so that interoperability can occur. Adaptation has received a significant attention in different areas including software component integration (e.g., [30, 29, 6]), process integration ([20]), and recently in the Web services area [22, 12, 11, 17, 7, 3]. It has been also accepted as a common practice to facilitate interoperation of heterogeneous applications in commercial products, e.g., in BEA WebLogic Adapters [2], and IBM WebSphere Integration Developer (WID)[15].

In the literature, many approaches (e.g., [11, 17], including some by the authors, e.g., [3]) attack the problem by identifying possible classes of mismatches between service interfaces and protocols and by suggesting ways to resolve mismatches in each class. As an example, we present below some of the most common mismatch classes [3]. We denote by *SP* the service provider and *SC* the service clients to be adapted:

- *message signature*. Message m in *SP* (corresponding to the request of a certain functionality[1]) has a different name and/or data types in the interface of *SC*.
- *message split/merge*. Message m in *SP* corresponds to (can be invoked by combining) messages $m_1, m_2, ..., m_n$ in *SC*, or vice versa.
- *missing/extra messages*. One or more messages in *SP* do not have any correspondence in *SC*, or vice versa.
- *message ordering*. The protocol definition of *SP* may expect a message m in a different order with respect to what sent by *SC*, or vice versa.

There are two subtypes of ordering mismatch: *unspecified reception*, in which one party sends a message while the other is not expecting it; and *deadlock*, i.e.,

[1] Receiving (sending) a message corresponds to invoking an operation (its reply, respectively).

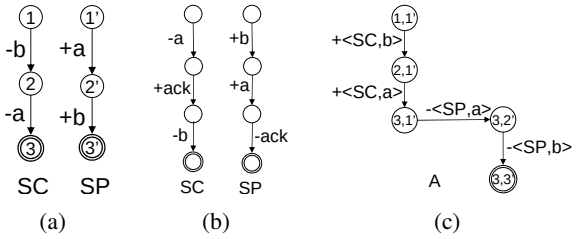


Figure 1: Ordering Mismatches: (a) unspecified reception, (b) deadlock, (c) an adapter for protocols in (a)

the case where both parties are waiting to receive some message from the other. To illustrate the concepts, consider the protocols of *SP* and *SC* in Figure 1(a): *SC* sends message `b` (shown by a `-b`), while *SP* does not expect to receive it (unspecified reception). In Figure 1(b) instead, *SC* expects to receive message `ack` after sending `a` (shown by `+ack`), while *SP* is waiting to receive `b` (`+b`). This is a deadlock case.

Adaptation in case of unspecified reception could be automatically handled as an adapter for protocols in Figure 1(a) can receive `b`, buffer it and send it to *SP* after exchanging `a` (e.g. see [30]). Figure 1(c) shows the adapter for protocols in Figure 1(a), in which $+\langle SC, b\rangle$ means adapter receives message `b` from service *SC*. However, adaptation in a deadlock case is a challenging task and requires extra knowledge (e.g., construction of messages `ack` or `b` in the adapter) to resolve the deadlock.

While identifying classes of possible mismatches between service specifications is important (as studied in [11, 17, 3]), the problem of service adaptation is not really addressed until we can assist developers in comparing two services, identifying which types of mismatches there are, and in developing the adapter. Approaches for automatic adapter generations for software component models [30, 6] and service protocols [7] exist. While they do provide interesting insights into the problem, they make the following assumptions regarding two key issues: (i) they assume there is no mismatch at the interface-level, or the interface mappings have been provided by the developer, and (ii) if there are interactions which lead to deadlocks, such interactions are considered as not adaptable. However, our experiments show that: first, the interactions of many real-world services may result in deadlocks; and second, careful analysis of some of such cases reveals that they are in fact adaptable (see e.g., the real-world example in Section 2).

In this paper, we present a method and tool that provides semi-automated support to mismatch identification and adapter generation. We aim at identifying and resolution of both interface-level and protocol-level mismatches and at providing a platform that can generate adapters semi-automatically and have them refined by the user in terms of deadlock resolution to generate the final adapter specification. Specifically, we make the following contributions in this paper:

- We provide a model for service adapters, which consists of interface map-

pings, and the adapter protocol. The adapter protocol represents the message exchanges of adapter with adapted services, and actions that instruct the adapter how to utilize interface mappings before/after each message exchange (Section 3.1).

- We provide semi-automated support to identify interface-level mismatches and identify the input for mapping functions that resolve those mismatches. We do this by leveraging approaches in XML schema matching [23], but we refine and extend them by considering, beyond message types, the contextual information provided by the service schema (the WSDL document). This enables a significant increase in precision for mismatch detection and resolution (Section 4).

- We provide automated support for identification of protocol-level mismatches, and generate adapter, if there is no deadlock. In addition, and most importantly, we propose a way to handle deadlock situations. We generate a tree, called *mismatch tree* for all mismatches that result in a deadlock. A mismatch tree provides a concise representation of all deadlocks and messages involved in each deadlock. Then, we make suggestions to resolve each deadlock by analyzing service interfaces, protocols and execution logs, if available. The combination of the concise tree representation and the suggestions for deadlock resolution assist the user in the decision makings leading to the generation of the final adapter (Section 5).

- We present an implementation of the approach in a tool, which assists users in the process of interface mappings, mismatch tree generation and analysis, and generation of adapter specifications. The tool has been implemented inside IBM WID (WebSphere Integration Developer) and in the context of Wombat project [19]. We experimentally validated our approach in both synthetic and real-world scenarios (Section 6).

Finally, in Section 7 we discuss related work and present the concluding remarks.

2 A Motivating Example

As a motivating example, we consider an adaptation task for services in the management of shopping carts. *XWebCheckOut*[2] and *Google Checkout*[3] are commercial checkout services. They provide a facility for sellers to manage the orders that they receive on their own websites. The only major difference between these two services is that *Google Checkout* also provides an administration website for buyers (people who do shopping on sellers' websites). Buyers register their details

[2] www.xwebservices.com/Web_Services/XWebCheckOut/

[3] code.google.com/apis/checkout/

with *Google* and manage their orders through that website. In *XWebCheckOut*, sellers provide administration support for buyers in sellers' websites. Some APIs are provided by *XWebCheckOut* to facilitate this task, for which there is no counterpart in *Google APIs*. Other than this, the two services offer similar functionalities, but through different interfaces and protocols.

Assume that *XWebCheckOutClient* (for short *CO_Client*) is a seller and a client of *XWebCheckOut*. For some reason (e.g., *XWebCheckOut* rises service fees), the client decides to either replace *XWebCheckOut* or extend its offering with *Google Checkout APIs*. Ideally, *CO_Client* would like to adapt its implementation to interact with *Google Checkout*, as opposed to developing a new client from scratch (see Figure 2).

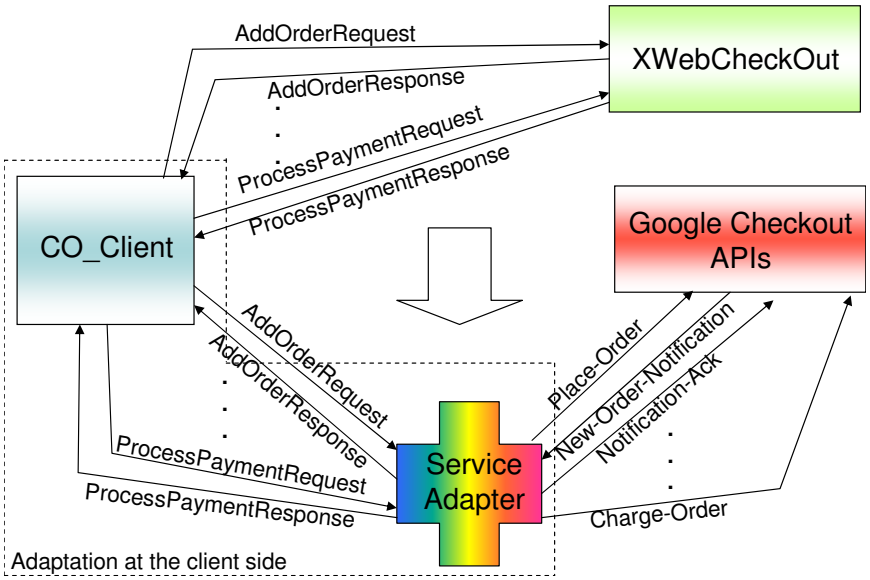


Figure 2: *CO_Client* to replace *XWebCehckout* service with *Google checkout APIs* using adapters

These two services provide similar APIs for order creation and management, payment processing, and order cancellation. However, there are differences in the interface definition (message names, number, and types) and how they exchange messages to fulfill a functionality. For example, Figure 3 shows the protocols of the two services for placing an order. Using existing approaches to adaptation, besides the problem of having to derive interface mappings "by hand", it would not be possible to derive adapters for these protocols. This is because their interaction results in deadlock, as in states 2 the *CO_Client* service expects to receive message `AddOrderResponse`, which is not supported by *Google*, while *Google* expects the message `Notification-Acknowledgment` in state *iv*. Hence, their interaction leads to deadlock. However, the services are in fact adaptable (by construction of above two messages in the adapter). In the following we show how

both the interface and protocol adaptation problems can be addressed in this example and in general.

3 Adapter Generation Principles

In this section, we first present a definition for service adapters, and then we give an overview of our approach in this paper.

3.1 Service Adapters

We introduce concepts and definitions to provide a formal basis to service adaptation. We begin with the interface definition, which is essentially a simple formalization of WSDL. An interface I of a Web service SP, denoted by I_s, is defined as follows:

Definition 3.1. *An interface I_s is a triplet $P = (D, M, O)$, where D is the set of (XML) data types of the service, O is the set of operations supported by the service, M is the set of messages exchanged as part of operation invocations, in which:*

- *a message m has optionally $i \geq 1$ parts, represented as $m =< d_1, d_2, ..., d_i >$, $m \in M, d_j \in D, 1 \leq j \leq i$*
- *$o =< m_{req}, m_{res}, m_f >$, that is, $o \in O$ is an operation associated to at least a request message m_{req} or to a response message m_{res} (or both) and possibly a fault message m_f.*

Next, now we extend the notion of mapping between component interfaces in [30] for Web services.

Definition 3.2. *Given interfaces $I_s = (D_s, M_s, O_s)$ of service SP and $I_c = (D_c, M_c, O_c)$ of service SC, an interface mapping $IM_{<s,c>}$ from SP to SC is a set of functions such that: $m \leftarrow func(X)$, $m \in M_s$ and where the input X is either a set of messages $\{m' | m' \in M_c\}$, or a constant value, or an empty set.*

The interface mapping $IM_{<s,c>}$ may contain more than one mapping functions for a given message $m \in I_s$, or may not contain any function for another message $m' \in I_s$. This definition allows for specifying $1-1$ mappings (to resolve message signature mismatches) and $1-n$ mappings (resolving message split/merge mismatches) between messages of the two interfaces. Based on messages mappings in $IM_{<s,c>}$ we can establish the mappings between operations O_s and O_c in I_s and I_c. Finally, we define the the interface mapping IM_A for the adapter as the union of mappings from interface I_s to I_c, and from I_c to I_s, that is $IM_A = IM_{<s,c>} \cup IM_{<c,s>}$. We use *im* to refer to a given mapping function for message m in IM_A.

We adopt finite state machines (FSM) as the modeling formalism for business protocols [14, 4]. FSM is a well-known paradigm, easy to understand and formalize for developers, and widely used for modeling business interactions [8, 18].

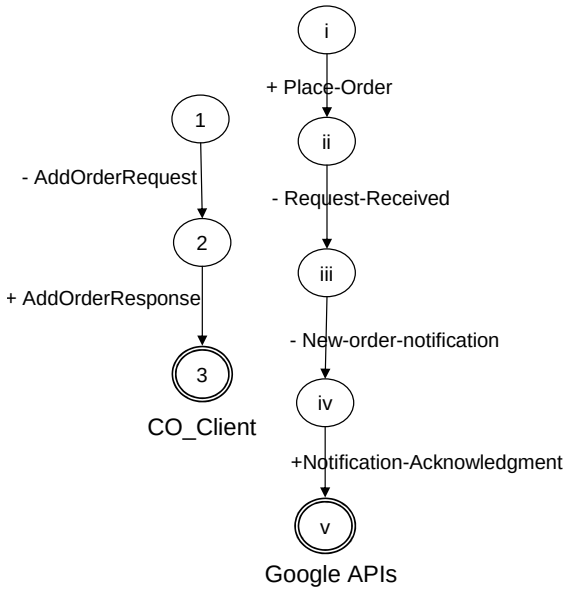


Figure 3: The detailed protocols of *CO_Client* and *Google checkout APIs* for placing an order

Finally, it is a quite familiar notation for developers by resemblance to UML state diagrams and UML state charts commonly used for modeling software business processes, and it is also supported inside IBM WID.

Definition 3.3. *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.*

We define the notion of adapter for service protocols by extending the proposal of [30] for software components as follows. An adapter is analogous to protocol model where states are pairs of states of the services to be adapted, transitions are labeled with a message along with the message target (*SP* or *SC*). In addition, adapters have *actions*. Actions are associated to transitions and allow adapters to, for example, store messages (to handle ordering mismatches) or to apply message transformations by utilizing mapping functions.

Definition 3.4. *The protocol of an adapter A (denoted by P_A) for adapting interactions between P_s and P_c is a protocol with the following extensions:*

- $M_A = M_s \cup M_c$.
- *Each state s_A of P_A is a pair $\langle s_s, s_c \rangle$, in which s_s (s_c,respectively) indicates the corresponding state of P_s (P_c, respectively) while adapter is in state s_A.*

- *Each transition t_A of adapter is shown in form of $+/-\langle s_A, s'_A, partner, m\rangle$, in which $+$ (or $-$) specify that the adapter A is receiving (sending, respectively) message m from (to) partner service, and takes the adapter from state s_A to s'_A. The parameter partner can be one of SP or SC.*

- *A transition t_A may be associated to actions "save (m)" and "activate (m)" after receiving a message m; to actions "synthesize (m,im)" , $im \in IM_A$ before sending a message m, and to action "inactivate (m)" after sending a message m.*

The optional actions allow to instruct the adapter to use interface mappings information with each transition. $save(m)$ instructs the adapter to save message m inside the adapter. For each message $m \in M_A$, an activity flag is kept inside the adapter, and $activate(m)$ ($inactivate(m)$) actions sets/unsets the activity flag of a message m to true in the adapter to show if the adapter has received the message. Finally, $Synthesize(m,im)$ instructs the adapter to use the interface mapping im information to construct m.

However, to generate the required actions associated to each transition in the adapter, it is not enough to only have interface mappings. This is because interface mappings do not contain the information on which point in the interactions and on how to use a given mapping. For example, we may like to enforce that for a given message m to use mapping im_1 in one part of interactions, and im_2 in another part. So, we need to define rules that governs where and how to use a specific interface mappings. The general forms of rules is as follows:

Definition 3.5. *A ruleset $R_A = \{r_1, r_2, ..., r_n\}$, $n > 0$ is a set of rules r in the form of $\langle m, im,$ activation-type, usage-type$\rangle$. The activation-type specifies at which point in interaction a given mapping for a message is synthesized and activated. This field can take one of the following values:*

- *on-reception: activate m on receiving it. This is the default value.*

- *before-sending: use the interface mapping im to synthesize m before sending it.*

- *in-state $\langle s_s, s_c\rangle$: use interface mapping im to to synthesize m while in state $\langle s_s, s_c\rangle$.*

The usage-type specifies how many time to use a given mapping or message before inactivating it. It takes one of the following values:

- *zero-times: message m should not be used after it is received (it is an extra message).*

- *one-time: use interface mapping im to synthesize m once and then inactivate it. This is the default value.*

- *unlimited: never inactivate a message m synthesized using mapping im.*
- *after-sending m': after message m' has been sent, message m is inactivated.*
- *in-state $\langle s_s, s_c \rangle$: message is inactivated in state $\langle s_s, s_c \rangle$.*

Definition of rules was first introduced in [30]. Here, we made two contributions in addition to this work: (i) we extended the language for rule definition, as we could not define some rules like "invalidate m after sending m' or in state $\langle s_s, s_c \rangle$ as a part of interface rules, (ii) we abstracted the rules away from the details of interface mappings and also adapter specifications, so rules can be defined separately and supplied to the adapter generation algorithm. This simplifies the job of the developers in defining rules.

Finally, given the above definitions, an adapter is defined as follows:

Definition 3.6. *An adapter A for protocols P_s and P_c is specified with a tuple $A=(P_A, IM_A)$.*

To conclude this section, an adapter A is specified with the protocol of the adapter (P_A) and the set of interface mappings IM_A, and the set of actions associated to each transition in the adapter that specifies how to use interface mappings. The set of rules R_A are needed during the adapter generation phase to help in generation of actions, however, rules are not part of adapter specification.

As discussed before, unlike existing approaches for automated adapter generation in [30, 7, 6], we do not assume the the interface mapping IM_A is provided, but we propose an approach to help the developer in providing interface mappings. In our approach interface mappings is performed in a two-step process: (i) *identifying interface matching*, which is the process of identifying the relationships between messages in I_s and I_c. This includes identifying relationships between the data types of messages in the two interfaces. The purpose of this step is to find the set X of parameters of the function $func(X)$ that generates m; (ii) *Specifying mapping functions*. In this step, the mapping function $func(X)$ that returns m is specified. We propose a methodology to help the user in performing the first step, as discussed in Section 4. The second step is performed by the adapter developer, as discussed in Section 6. The identification of the matching between data types of messages in X and message m is the most important part in specifying $func(X)$.

3.2 Approach Overview

Adapting heterogeneous services is generally a hard problem. We approach it by trying to "'encode"' the approach humans take when developing adapters by hand. We do this by trying to manually adapt service protocols in the Google checkout example and by recording our reasoning principles and methodology. We make the following observations in such a practice, which constitute the building blocks for the process of adapter generation we propose, as depicted in Figure 9:

1. The adaptation process starts by finding the correspondence between messages of services. In terms of adaptation, this translates into identifying mismatches of types of message signature, split/merge, and extra/missing.

2. Once message mappings are known, experts would examine all possible message exchanges between two services based on protocol definitions. During this phase, all protocol-level mismatches will be identified: (i) finding all ordering mismatches of type unspecified receptions between protocols, and generating the adapter specification for them; (ii) identifying all interactions that result in deadlocks and messages that are engaged in deadlock.

3. For each deadlock, experts try to gather some *evidence* (see below) to conclude if the deadlock can be resolved by construction of messages that are responsible for the deadlock. If such evidence is found, then the corresponding interface mappings are added to the set of plausible interface mappings, and the corresponding rules are generated.

4. Evidence ranges from one or some of the following, which most rely on analyzing interface and protocol specifications of services: (i) inspecting the content of required messages by looking at WSDL/Schema definition. In some cases, the required message, typically of type of acknowledgment or response messages, are of empty contents, or the expected values for various elements in the message is given through an enumeration. (ii) experts may look at the log of previous interactions of the services (or the client, depending on which side the adapter is generated), if available, to see which data/values are exchanged for the required elements. (iii) experts may also consider the service protocol and the interface (or those of the client, depending on which side the adapter is generated) to see if some other operation(s) could be invoked, i.e., their input is available in the adapter, for which their outputs provide the required elements of data.

4 Interface-Level Mismatches

Given two service interfaces I_s and I_c and protocols P_s and P_c, the goal is to find the matching between messages in interfaces I_s and I_c. In this phase, we do consider not only the information in I_s and I_c, but also the ordering constraints that protocols define. We argue that interface matching cannot be addressed properly without considering the ordering constraints, as well, since e.g., a given mapping function $m \leftarrow func(m_1, m_2)$ may seem possible by looking at the interface-level information, but considering the protocol information, the adapter may not have received m_1 and m_2 when it is needed to synthesize m. In the following, we present a semi-automated approach for identifying a set of initial matchings based on information at the interface-level, and then discuss in Section 4.2 how we improve the matching results based on protocol-level information.

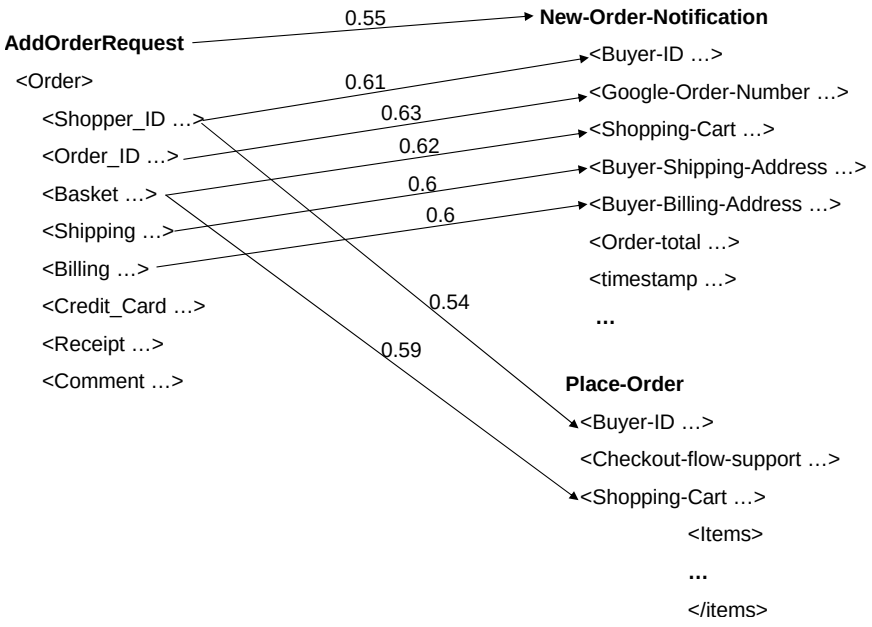


Figure 4: `AddOrderRequest` and its candidates for matching in *Google checkout APIs*

4.1 Interface Matching

As mentioned before, we base our interface matching on approaches in schema matching [23]. The reason is that like approaches in schema matching, we are interested in finding the matching between the data elements in the schemas of two services. This helps us to find the relationships between messages of Web services. Schema matching is a hard problem. In general the results on large and arbitrary schemas (of services) may not be always useful [24]. Fortunately, we have additional information compared to schema matching approaches in service interfaces, which are message and operation definitions, that act as additional constraints. We use the following heuristics based on message and operation definitions to increase the precision of matching in the matching of schema definitions of any two service interfaces:

Pair-wise matching of schemas of messages. Our experiments with schema matchers show that usually we do not get precise matching results using the whole schemas of two services at once. Working on service (WSDL) interface allows us to break down the problem of schema matching into matching schemas of individual messages of two schemas. We identify fragments of schemas to be compared at each step. We perform such comparison for all pairs of messages from the two interfaces. This results in increasing the precision of each matching, however, the number of required matchings increases from one to the Cartesian product of number of messages in the two interfaces. We believe that is an acceptable overhead to achieve higher precision.

To illustrate the approach, let's consider the schema of <Order> in *XWebCheckout* and its corresponding matches in *Google APIs* depicted in Figure 4. We used COMA++ [9][4] to find the matching between the whole schema of *XWebCheckout* and that of *Google Checkout*. The only matched elements where <address> in <Billing> and <Shipping> to <address> data type in *Google*. However, in pair-wise comparison of <Order> (schema of AddOrderRequest message) the result was more precise. The relationships between messages Place-Order and New-Order-Notification are captured as depicted in Figure 4. These two schemas are the closest to <Order> in the schema of *Google Checkout* with the matching score of 0.29 and 0.55, respectively. This is because the parameter of AddOrderRequest is of type Order, which has the following schema elements: Order_ID, Shopper_ID, Basket, Shipping, Billing, Credit_Card, Receipt, and comments (Figure 4). Basket contains all items that are ordered by buyers. Shipping and Billing specify the shipping and billing addresses, respectively. In *Google*, new-order-notification contains almost all the data types in Order, however, Order and Place-Order are matched only in Shopper_ID, Buyer-ID, and Basket, Shopping-Cart.

Finally, we used the observation that if some parts of a message are matched with elements in one message and some other parts of the same message are matched with elements of other messages, then it is an indication of a merge/split mismatch (1-n matching).

Incorporating message name into the schema. Our experiments also show that if we incorporate the message name into the schema for that message, it increases the precision of mappings. This is performed through creating a new complex XML element, named after the message, and includes the schema of the messages. This is considered in Figure 4.

Considering the message type. An indication that helps in reducing the number of required pair-wise message matchings is considering the message type, i.e., if a message is an input or output of an operation definition. When generating adapters for compatibility (adapting a client to work with a given service), we only check the matching between output (input) messages of each operation of the client interface with the input (output, respectively) messages in the service interface. When generating adapters for replaceability (developing the adapter to make the specification of a service similar to another given service) we check only matching between the input (output) messages of operations of a service with the input (output, respectively) in the other service interfaces.

In Figure 4 we have the interfaces of the two services *XWebCheckout* and the *Google Checkout*. Without considering the operation definitions, the matching results in Figure 4 suggest that message new-order-notification is the best match for message AddOrderRequest based on the matching score. However, if we consider the operation definition constraints on mappings, we observe that AddOrderRequest is an input message for AddOrder operation, while mes-

[4]available at dbs.uni-leipzig.de/de/Research/coma.html COMA++ is one of the best available schema matchers that enjoys from combining several available methods for schema matching

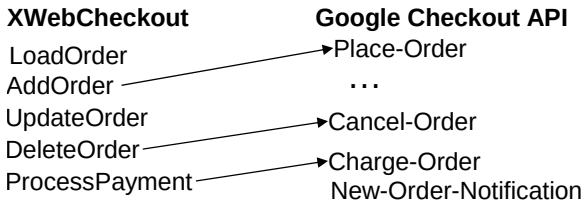


Figure 5: Operation mapping between *XWebChecoutClient* and *Google checkout APIs*

sage `new-order-notification` is an output message in a notification operation with the same name. On the other hand, `Place-Order` message is the input of `Place-Order` operation. So, considering the operation definitions we conclude that the only `AddOrderRequest` is a possible match for `Place-order`, although it has a smaller matching score.

The following algorithm summarizes our interface matching method, in which I_1 and I_2 denote the WSDL interface of two services:

Algorithm 1 Interface Matching Algorithm

Require: I_1, I_2
Ensure: Message Matching between I_1, I_2
1: $XSD_m \leftarrow$ XML schema of message m in I_1 (I_2)
2: **for** message $m \in I_1$ **do**
3: **for** message $m' \in I_2$ **do**
4: match($XSD_m, XSD_{m'}$) considering message types (input/output)
5: **end for**
6: **end for**
7: Perform 1-n message matching

Figure 5 shows the matches for some of the operations in the two interfaces for *Google* and *XWebCheckout*. The result of matching indicates all operations required by *CO_Client* are covered by *Google* except two operations, which are `LoadOrder` and `UpdateOrder`. In fact, these two operations are used by *CO_Client* in its Website to allow buyers to load and update orders. However, since *Google* provides a separate website for buyers, these two operations are not needed to be invoked. On the other hand, there are many messages in the *Google* interface that do not have a match in *CO_Client*, e.g., `new-order-notification`. This is an extra message in the *Google* interface.

4.2 Applying Ordering Constraints

As discussed before, it may not be always possible to use all matching results that are generated based only on the interface information during the adapter generation. This becomes clear by considering the ordering constraints that two services impose on the exchange of messages. We refer to such matches as *non-*

plausible matches. As an example, let's consider the protocol definitions `SP` and `SC` in Figure 1(b). Interface matching results for these protocols specifies that message <*SC*, `ack`> is matched to message <*SP*, `ack`>. However, considering ordering constraints, we observe that the message <*SP*, `ack`> is not received by the time that <*SC*, `ack`> is needed, so this mapping is not plausible. So the mapping function of message <*SC*, `ack`> cannot take the set X identified in this step.

So, we need to verify the interface matchings generated based on the information level information, and identify the set of not plausible interface mappings using the ordering constraints defined in protocols. This has to be done before proceeding to ask the user to generate the interface mapping functions that take the input X and transform it to message m, as otherwise such mapping functions will be useless. However, there may be other possible matching (e.g., a different set X) that makes the mapping possible.

Among all types of possible mismatches that we studied in the paper, decision making regarding if we need to develop mapping functions for extra/missing messages in the two interfaces also could not be answered without considering the protocol-level information. This is because the protocol-level information will clarify *if* and *at what state* during the interactions such messages are required. For example, there are many messages in the *Google* interface, e.g., message `merchant-calculation-results`, which is not communicated with *CO_Client*, and also `New-Order-Notification`, which is sent as a part of placing order but client does not require it. However, `Notification-acknowledgment` should be provided for a successful interaction with *Google*, so a mapping is required to be provided. Answering all of these questions requires protocol-level analysis. In the next section, we present our approach for providing above mentioned analysis.

5 Protocol-level Mismatches

After applying the interface matching techniques, we get the matching between messages of I_s and I_c, and so the interface mapping IM_A. Given IM_A, in this section, we use the protocol definitions P_s and P_c of the two services to find all protocol-level mismatches. As discussed before, there are two types of mismatches at the protocol-level: unspecified reception, and deadlock. Handling unspecified reception mismatches could be performed automatically (see e.g., [30, 6, 7]). Thus we do not discuss this aspect further. However, handling mismatches with deadlock is challenging, and has not been addressed before. As we explain in the following, this type of mismatch is caused by either the lack of required interface mappings in IM_A, or non-plausible interface mappings in IM_A. In fact, we only understand these two cases by considering the protocol-level information. In this section, we propose techniques to identify such mismatches, represent and analyze them to help the user in providing new interface mappings in IM_A or refining existing interface mappings in IM_A, if possible, to avoid the deadlocks during adapter generation process.

We perform protocol-level analysis through simulating the adapter generation process, which explores all possible message exchanges between the two services according to P_s and P_c. So, in the following we first present an algorithm for adapter simulation. Then we discuss our approach for providing a concise representation of all mismatches with deadlock, and analyzing them to make suggestions to the developer to resolve such mismatches. Suggestions are in form of identifying the input X for the mapping functions that enable construction of the messages that are engaged in each deadlock.

5.1 Adapter Simulation

The following algorithm (Algorithm 2) simulates the interactions between two protocols in an adapter starting from a given state of the adapter (e.g., $\langle s_{pinit}, s_{sinit} \rangle$, which corresponds to the initial states of protocols P_s and P_c). The input of the algorithm is protocols P_s, P_c, and interface mapping IM_A between I_s and I_c. The output of this algorithm is protocol P_A. It also identified deadlock cases, and allow for handling them (variables MT and $StackInfo$ are introduced in the next sections).

Algorithm 2 SimulateInteraction

Require: $StackInfo, P_s, P_c, IM_A$
Ensure: P_A, MT
1: $Q \leftarrow StackInfo.Q$
2: **while** $Q \neq \varnothing$ **do**
3: $s_A \leftarrow dequeue(Q)$
4: $Out \leftarrow FALSE$
5: **if** $(s'_A, t'_A) \leftarrow TransitionOut(s_A, P_s)$ & $s'_A \neq s_A$ **then**
6: $enqueue(Q, s'_A)$;
7: $AddTo(P_A, s'_A, t'_A)$
8: $Out \leftarrow TRUE$
9: **end if**
10: **if** $(s'_A, t'_A) \leftarrow TransitionOut(s_A, P_c)$ & $s'_A \neq s_A$ **then**
11: $enqueue(Q, s'_A)$;
12: $AddTo(P_A, s'_A, t'_A)$
13: $Out \leftarrow TRUE$
14: **end if**
15: **if** $Out == FALSE$ **then**
16: $identifyNonPlausibleMappings()$
17: $handleDeadlockState(s_A, stackInfo.MTcurrentState)$
18: **end if**
19: **end while**

The variable Q implements a queue structure that is a list to keep track of all possible state s_A of the adapter. For each s_A, function $TransitionOut(s_A, P_x)$, $P_x = P_s$, or $P_x = P_c$, checks if there are possible message exchanges between the service and the adapter, or the state s_A is a deadlock state ($Out == False$). A state

is a deadlock state if there is no possible message exchange between the service and any of the two services *SP* and *SC* in that state. This happens if both services are waiting to receive some messages that the adapter can not synthesize. If one of the following two conditions holds it means that a transition out of s_A exists and the adapter transits from state s_A to s'_A:

- P_x is ready to send a message, so the adapter receives it, generates s'_A, and corresponding $t'A$, and the set of actions for $t'A$ (saving the message in the adapter, and activating its flag).
- P_x is ready to receive a message m that is associated to a mapping $im : m = func(m'_1, ..., m'_k)$, and all input messages $m'_1, ..., m'_k$, $k > 0$ are received in the adapter (their activation flags are equal to True). So, the adapter generates s'_A, and corresponding $t'A$, and the actions to synthesize m.

If outgoing transitions are found for a state s_A, then the new states of s'_A are put in Q, and s'_A and corresponding t'_A are added to P_A. Otherwise, s_A is a deadlock state. In existing approaches for automatic adapter generation [30, 6, 7], deadlock states are removed from the adapter, i.e., such interactions are not supported by the adapter. However, in the following we propose an approach to give the user an opportunity to examine if the adaptation is possible or not (line 17 of Algorithm 2), which is detailed in the next section). The complete adapter simulation process is presented in Section 5.3, after all necessary procedures are introduced in the following.

5.2 Handling Mismatches with Deadlock

As discussed before, mismatches with deadlock are due to one of the following two cases: (i) provided mapping *im* for a message *m* is non-plausible, or (ii) there is no mapping provided for a message *m* that is required during the interactions (e.g., for a missing/extra message). To illustrate the approach, let's consider protocols *SP* and *SC* in Figure 6(a), in which their interactions result in deadlock as both are waiting to receive some messages in states 1 and $1'$, respectively (*SP* is waiting for message `a` and *SC* for message *c*). We propose the following two approaches for dealing with such situations: (i) progressive user interaction, (ii) mismatch tree generation.

5.2.1 Progressive User Interaction

In this approach, as soon as the algorithm finds a deadlock state (line 17 of Algorithm 2), we prompt the user with messages that are responsible for the deadlock. For example, for protocols *SP* and *SC* in Figure 6(a), when adapter is in state $\langle 1, 1' \rangle$, we prompt the user and ask for the mappings function for one of messages $\langle SP, a \rangle$ or $\langle SC, c \rangle$ to resolve the deadlock. To facilitate decision making for the developer, we provide information on how it might be possible to construct each

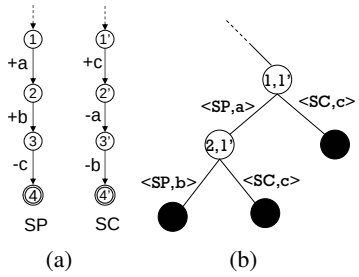


Figure 6: (a) ordering mismatch with deadlock (b) the *mismatch tree* for *SP* and *SC*

of these messages based on available evidences (See section 5.3.1). The developer may confirm that it is feasible to provide a mapping for one of these messages. In this case, the process of adapter generation proceeds until finding the next deadlock state. However, the developer may acknowledge that no mappings could be provided for any of these messages, then this deadlock state is tagged to be removed from the adapter during the adapter generation.

This approach is simple, however, it has two main disadvantages: (i) it may involve too many interactions with the developer, (ii) more importantly, as the future message exchanges of two protocols after the deadlock point is not taken into the account, the developer may not make the best decision. For example, assuming that for resolving the deadlock in Figure 6(a) it is possible to provide mapping functions for any of $\langle SP, a\rangle$ or $\langle SC, c\rangle$ and the developer selects to provide for $\langle SP, a\rangle$, then the next deadlock occurs between $\langle SP, b\rangle$ and $\langle SC, c\rangle$. However, if the developer had decided to provide mappings for $\langle SC, c\rangle$, then no more deadlocks would have occurred.

5.2.2 Mismatch Tree Generation

Motivated by the goal of providing a finding all deadlock cases between two protocols, in this approach, we perform a what-if analysis for each deadlock case, in the sense that: assuming that a mapping could be provided for each of the messages engaged in the deadlock, then how the message exchanges between two protocols proceed until the exchange ends up in final states in both protocols. Based on the result of this analysis, we build a tree, which is called a *mismatch tree* (MT). A MT represents all possible deadlocks between two protocols, and the messages that are engaged in each deadlock. For example, the MT for *SP* and *SC* in Figure 6(a) is depicted in Figure 6(b). It states that in state $\langle 1, 1'\rangle$ of adapter (state 1 of *SP* and $1'$ of *SC*, respectively), there is a deadlock that messages `<SP,a>` and `<SC,c>` are involved in it. From the deadlock resolution point of view, this node represents a choice (or-condition), in which if a mapping for either of `<SP,a>` or `<SC,c>` is provided the deadlock is resolved. It also shows all future deadlocks that would occur in each path of the tree. For example, if the developer provides a mapping

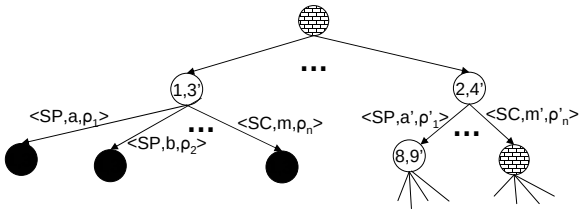


Figure 7: The general representation of a mismatch tree. Shaded nodes specify *and* nodes, and black nodes the leaves of the tree

for `<SP,a>`, then the next deadlock occurs in state $\langle 2, 1' \rangle$, which represents a choice between `<SP,b>` and `<SC,c>`.

In general, *MT* is an *AND-OR* tree. An *AND-Node* does not represent a deadlock case itself, but specifies that all deadlock cases that are children of this node should be resolved. The root of *MT* is an example of an *AND-Node*. The outgoing edges of an *AND-Node* do not have any label, but they are linked to other *AND-Node* or *OR-Node*. On the other hand, an *OR-Node* refers to a deadlock case. This type of node has one outgoing edge corresponding to each message that is engaged in the deadlock. It has one edge in the case that one of the services *SP* or *SC* is in a final state and the other requires the message on the label of the edge. Otherwise, it has more than one edges that specifies at least one of the messages should be provided to resolve the deadlock cases. The label of an *OR-Node* consists of the name of a pair of states of the service and the client, in which a deadlock occurs. The label of outgoing edges of an *OR-Node* represent the name of messages that are engaged in the deadlock. See Figure 7 for the general representation of a *MT*, in which both nodes of types *AND-Node* and *OR-Node* are present. In this perspective, *MT* in Figure 6(b) is a subtree that shows the messages that are engaged in the deadlock that occurs in state $\langle 1, 1' \rangle$ of the adapter. The advantage of *MT* is that it represents all possible deadlocks in a concise form, and allows the developer to make informed decisions.

Algorithm 3 shows the process for handling deadlock states in the adapter. For each deadlock state, since it is one of the possible deadlocks, we create a leaf node in *MT* for an *AND-Node* (line 1) (e.g., when $currentMTstate$ is its initial state). This is to specify that all the deadlock cases that occur should be resolved. Then, for the present deadlock cases, it checks the service and the client states from the adapter state s_A. If the service (the client, respectively) is not in a final state, then it calls `updateMT` for updating *MT* to create required nodes to show messages that are engaged in the deadlock.

To perform the what-if analysis for each deadlock case, we need to keep track of deadlock states, and also keep information regarding the messages that are engaged in the deadlock. To implement this idea, we use structure called `StackInfo`, and inserting it inside a stack. In the next algorithm 5 we show how this information is used to perform the what-if analysis.

Algorithm 4 shows how *MT* is updated for each deadlock case, and also the

Algorithm 3 handleDeadlockState

Require: $s_A, currentMTState$
Ensure: Revised MT
1: $newMTstate \leftarrow$ `createLeafInAND-NODE(currentMTstate,` s_A`)`
2: **if** $!(s_A.ServiceStateisFinalState)$ **then**
3: $updateMT(s_A, newMTstate, s_A.ServiceState)$;
4: **end if**
5: **if** $!(s_A.ClientStateisFinalState)$ **then**
6: $updateMT(s_A, newMTstate, s_A.ClientState)$;
7: **end if**

corresponding `StackInfo` for all messages that are engaged in each deadlock is built, and inserted into the stack. This algorithm creates an edge from $currentMTstate$ for each outgoing transition t from the $partyState$ (either client or the service state) that require a messages `msg` to fire (i.e., transitions having a message with a "+" as the label, e.g., `+msg`). All of these edges are children of an *OR-Node* to represent a choice to resolve the deadlock (line 2).

Algorithm 4 updateMT

Require: $s_A, currentMTstate, partyState$
Ensure: Revised $MT, Stack$
1: **for** each $t \in partyState.TransitionsrRequireMessages$ **do**
2: $newMTstate \leftarrow$ `createLeafInOR-NODE(currentMTstate,`s_A`, t.msg);`
3: $newQ.enquqe(s_A)$
4: $StackInfo.Q \leftarrow newQ$
5: $StackInfo.msg \leftarrow t.msg$
6: $StackInfo.MTstate \leftarrow newMTstate$
7: $Stack.push(StackInfo)$
8: **end for**

5.3 Complete Adapter Generation Process

Algorithm 5 shows the complete process of adapter simulation, and shows how the what-if analysis for each deadlock case is performed using the combination of a stack, and a queue structure. The behavior of the algorithm is captured in the following: (1) It generates the adapter specification for interactions that do not lead to deadlocks; (2) For interactions that lead into deadlock, and for each messages that are engaged in each deadlock case, it assumes that the messages could be constructed, then monitors how the interactions of the two service proceeds until the algorithm finishes in final states in both service and client protocols. The algorithm uses a queue to keep track of all states of the adapter to fulfill the first goal. It uses a stack structure to keep track of all deadlock states for the second goal. Each stack element, called `StackInfo`, contains the adapter state s_A which represents the state of the service and client protocols at the time that the deadlock happens, and also

the name of the message (`msg`) that should be assumed it could be constructed. Each element `StackInfo` (except the initial one) instruct the algorithm to assume that a given message `msg` is provided, and so the deadlock could be resolved by provision of message `msg`. Then, the adapter simulation process starts from state s_A (the input of *SimulateAdapter* is `StackInfo`), which was the deadlock state, and continues the adapter simulation process, which in turn updates the stack, MT, and the adapter specification P_A.

Algorithm 5 Adapter Simulation Process

Require: P_s, P_c, IM_A
Ensure: P_A, MT
1: $Q \leftarrow \{\langle init, init \rangle\}$
2: $StackInfo.MTcurrentState \leftarrow MT.initState$
3: $StackInfo.Q \leftarrow Q$
4: $Stack.push(StackInfo)$
5: **while** $Stack \neq \varnothing$ **do**
6: $\quad StackInfo \leftarrow Stack.pop()$
7: $\quad P_A, MT \leftarrow SimulateInteraction(StackInfo, P_s, P_c, IM_A)$
8: **end while**

5.3.1 Evidences

Determining if a given message m engaged in a deadlock could be constructed in the adapter or not is a very difficult task, and depends on many factors including the state in which the interaction between two services is, and also the semantics of messages. In the following, we discuss some of the evidences that can be used for identifying messages in common deadlocks appear in Web services interactions:

Interface-based inference. For a given message m from interface I, which is a label of an edge in MT, we can perform the following analysis on interface I to find indications that might help to construct m:

(i) *Messages with empty content*. By inspecting the schema of message m, we may observe that it is an empty message. This is specially the case for some acknowledgment and response messages.

(ii) *Analyzing the messages of the same interface*. If the data structure of a message m is not empty, then we analyze the relationship between data structure m, and those of all messages m_1, ..., m_k, $k > 0$, of the *same interface* I, that has been received before the deadlock point in the adapter. If elements of m could be matched to elements of any of above messages, we may be able to construct this message from those messages. This technique is also helpful on some response and acknowledgment messages that return some order-number, serial-number that is previously exchanged between services. In this case, the probability that m could be constructed is considered as the similarity score of elements of m to existing messages in the adapter from interface I.

(iii) *Enumeration with default*. In some schema definitions, e.g., in *Google*

APIs, the expected values for some data types are given through enumeration. It may be possible for the adapter to continue interactions with a service using some default values from such a list.

(iv) *Acquiring m through operation invocation*. In some cases, we may observe that message m from interface I_1 that is required in MT has a mapping to a message m' from the partner interface I_2. And, m' is the output message of operation $o\langle m'', m'\rangle$ with input message m''. And, we observe that we can construct m'' from the messages that already have been received by the adapter. This allow to get m through invoking operation o. The weight of m in MT is a product of matching score of m and m', and also the matching score of m'' to messages in the adapter.

Log based value/type inference. If the log of previous interactions of the service that we are developing the adapter in that service side is available, e.g., in case of *CO_Client*, it keeps the log of its previous interactions with *XWebCheckout*. Then, this log is used to infer the data types/values exchanged for specific elements in the required message m. For example, we may observe that a fixed value for data elements of m is exchanged, or it is part of previous messages exchanged between services. As another example, we used this evidence to adapt a client of version 1 of *XWebCheckout* service to use version 2 of this service. The main difference between these two versions is in the schema definitions: version 1 uses simple data types and all inputs are defined as strings, while in version 2 complex XML types are used to declare the expected schemas. Analysis of log of client made it clear that exchanged contents in messages of version 1 of the service are in XML format and conform to the XML schemas in the second version, with few exceptions. However, by considering only the schema definitions we could not make such an inference.

Developer Input. As discussed, determining if a given message m could be constructed in the adapter or not is very difficult and depends on many factors that may not be captured by any of above evidences. For this reason, we also rely on the input by the adapter developer to identify if it is feasible to provide a mapping function to construct a message m in the adapter or not.

5.3.2 Analyzing Mismatch Tree Using Evidences

As discussed before, we assign a weight to each edge in the tree based on the analysis of available evidences to show the probability that the message on the edge could be constructed. So, the complete representation of each message on edges of MT is in the format of $<P, m, \rho>$, in which P denotes the protocol name, m the message name, and ρ is the probability that message m can be constructed based on evidences Figure 7. This probability takes values between 0 and 1, in which value 0 specifies that we do not have any indication that this message could be constructed, while 1 suggests that this message could be constructed based on available evidences. At this stage, we end up with a weighted tree, in which each edge has a weight between 0 and 1. For each deadlock case in this weighted MT (i.e., for each subtree corresponding to one children of the root node of MT), we

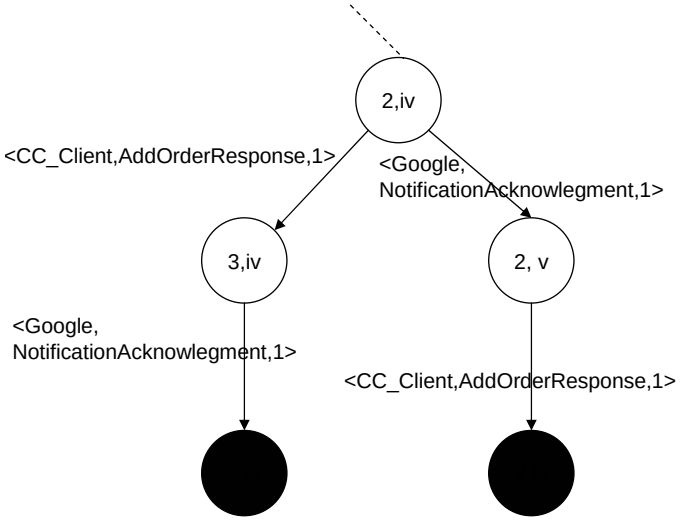


Figure 8: The mismatch tree for the placing order protocol of *CO_Client*, *Google Checkout* in Figure 3

are interested to find the shortest path, i.e., with the minimum number of messages, that maximizes the probability that the deadlock could be resolved by constructing messages that are engaged in the deadlock. The probability that we can construct one message is independent from the probability of the construction of any other messages. So, we can define the probability that each deadlock case is resolved using the set of messages in a specific path as the product of the weights of each edge in that path. Then, we rank different paths in each subtree (corresponding to each deadlock case). The result of this ranking is a list, in which the top path corresponds to the *best shortest* path in the weighted subtree, that we suggest to the adapter developer.

5.3.3 Mismatch Tree for the Running Example

Let's consider very simple protocols of *CO_Client* and *Google Checkout* depicted in Figure 3. In Figure 4 we concluded that `Place-Order` message is a plausible matching for `AddOrderRequest` message in *CO_Client*. However, there is no matchings for messages `New-order-notification` and `Request-Received` in *CO_Client*, and also no matching for `AddOrderResponse` in *Google Checkout*. Figure 8 shows the mismatch tree MT generated for these two protocols. The first mismatch with deadlock occurs in state $\langle 2, iv \rangle$ between messages `AddOrderResponse` and `Notification-Ack`. If a mapping for either of these messages could be provided, then there will be another deadlock case which shows the other message is still required (in states $\langle 3, iv \rangle$ and $\langle 2, v \rangle$). Since `Request-Received` and `New-order-notification` are of type of extra messages they do not create any deadlock, but the adapter could receive them. However, after receiving `Notification-acknowledgment`,

it needs to send message `Notification-Ack` to the *Google*. This has been captured by the mismatch tree.

To assign weight to different paths of MT in Figure 8, in the first step, we analyzed the WSDL interfaces of *CO_Client* and *Google Checkout*. `Notification-acknowledgment` has a `serial-number` element as its content. Considering the relationship between these messages and `New-order-notification` reveals that it is an element of this message that is sent by *Google Checkout*, and so we estimate that the probability of provision of this content as 1. In case of *CO_Client*, inspecting the content of message `AddOrderResponse` shows that it is an empty message. So, in this special case, the probability of construction of this messages is also estimated to be 1. Based on this analysis, the adapter developer can create the required mappings in IM to resolve the deadlock.

To summarize, the adapter simulation process in the algorithm 5 generate one of the following results for any given protocols P_s and P_c and interface mappings IM: (i) it outputs that there is no adaptation required, and protocols successfully interact (if there is no protocol-level mismatches); (ii) adaptation is required, but there is no deadlock interactions (there are mismatches of type of unspecified reception), and generates the adapter protocol (P_A); (iii) the adaptation is required, and there are deadlock interactions. For deadlock interactions, the mismatch tree MT is generated for further analysis (line 17). Based on analysis of MT the developer may updates interface mappings IM to resolve some deadlocks, or tag some of the deadlocks as non-resolvable. In the next run of the algorithm, it removes all the deadlock states tagged as non-resolvable.

6 Implementation and Experiments

The approach presented in the paper has been implemented inside IBM WID (WebSphere Integration Developer), which is an Eclipse-based IDE for development of composite applications based on SCA (Service Component Architecture) architecture [25], and in the context of Wombat project for analysis of service interactions [19]. Figure 9 shows the architecture of the tool. Services and the adapter have been implemented as service components in SCA, the interface mapping component uses `InterfaceMap` components in IBM WID to implement mappings. The mismatch tree editor is implemented by extending the state machine editor in WID to represent mismatch trees, and the backend to check for evidences. Mapping functions could be implemented as XQuery, XSLT or even plain Java functions. Interface mapping editor allows developers to create new and also edit discovered mappings. The interface matching component is implemented on top of COMA++ tool [9] (*http://dbs.uni-leipzig.de/de/Research/coma.html*).

Our tool also supports defining rules that specify how to use interface mappings during the adaptation. For example, if two or more mappings are specified for a given message m, rules specify where to use which one of the mappings, or to define generalized forms of actions of adapter. For example, by default after

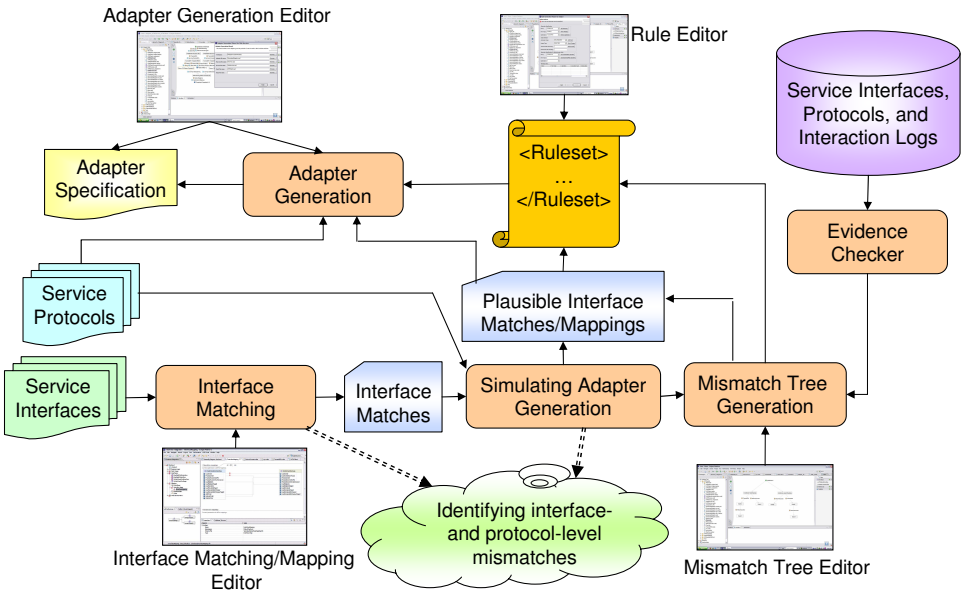


Figure 9: The architecture of our adapter development tool

sending a message m, all set of input messages in X that are used to build m and m itself is inactivated. However, the developer can define rules to allow a given mapping to be used in any state that is needed during adapter generation and not to be inactivated. Rules are represented in XML and implemented following the EMF (Eclipse Modeling Framework) in Eclipse. The rule editor allows the developers to edit discovered rules or create new rules for a pair of protocols. The adapter generation process accepts the rule definitions as the input of the algorithm. Finally, all algorithmic parts are implemented using Java 1.5.

The result of application of the tool on matching *CO_Client* (in fact *XWebCheckout*) and *Google Checkout APIs* for other functionalities (order payment, shipping, and cancellation) shows that the interactions of the two services results in deadlocks, and the mismatch tree generation and ranking approach is very useful in enabling the adaptation between the two services. In addition, we have applied the tool on a number of other service interfaces and business protocols taken from the real-world scenarios, e.g., an ATM/Bank interface and protocol definitions [19], mapping a client of version 1 of XWebChecout service to work with version 2 of the service and the interface and protocol definitions of a purchase order service taken from [1]. The lessons learned from these experiments include: (i) in many services, only a subset of elements of the schema defined for each message is essential for the proper functioning of the service. Functionality-wise similar services often declare such essential information in their interfaces. Our interface mapping methodology proved effective in finding the matching between messages, and between data types of each message for such parts; (ii) such functionality-wise similar services often define different ordering constraints on the message

exchange, and mainly the differences are of type of signature mismatch or having extra/missing messages in service interfaces. This later case often leads the interaction to deadlock.

7 Related Work and Conclusion

The problem of adapting interactions models in software has been studied in different contexts, and more notably in the area of software components (e.g., [30, 6, 29, 16]) and also recently in Web services [3, 12, 11, 17, 7]. There are mainly two schools of work in this area: The first school of approaches propose techniques for automatic generation of adapters [30, 6]. All of them tackle ordering mismatch with unspecified reception, and remove all interactions that lead to deadlock from the adapter (hence, deadlock situations are not in fact managed). In addition, they assume there is no mismatch at the interface level, or that interface mappings are provided. The other school of approaches provide classes of possible mismatches between interactions models and then propose adaptation templates based on design patterns or adaptation operations to resolve the mismatches (e.g., [13, 11, 17]). However, in all of these approaches developers need to manually inspect the protocols and identify the mismatches.

At the interface level there are two main approaches in the prior art: (i) approaches for finding similar operations in a repository of service descriptions like UDDI to a given textual description or to some service operation signature, e.g., [10, 27]. However, the objective of these approaches is not to find the exact mapping between elements of messages (and operation), but to find a measure of their similarity typically based on information retrieval techniques. In fact, the proposed techniques are not applicable in interface mapping context as we have only two service interfaces to map, while e.g., Woogle [10] proposes a clustering-based approach that requires a repository of service descriptions to be applied as a learning phase. The second class of prior art propose approaches for adapting a service WSDL interface to incompatible clients, e.g., [12, 22]. In [22] authors assume that interfaces of all services that provide a similar functionality are derived from a common base interface using limited number of derivation actions that allow for adding or removing parameters to operations, however the operation names remain the same. We do not make any assumptions on service interfaces. We build on top of schema mapping approaches [23, 9] and we extend them by considering protocol definitions to identify the set of relevant mappings in service interactions. In [12], the author proposes defining service views on top of WSDL interfaces by altering WSDL interfaces to enable interactions with incompatible services, but no automatic support for generation of views is proposed.

Semantic Web-based approaches based on ontologies also provide an attractive alternative for Web service matching (e.g., [1, 21]) and mediation [28]. However, the limited availability of ontologies in real-world Web services makes it hard to apply these techniques at this stage. Commercial products, e.g., IBM WID, BEA

WebLogic or Microsoft Biztalk also provide facilities for manual mediating between service interface and protocols, however, they offer limited automated support.

In summary, the innovative contributions of this paper lie in, first, providing automated support for identification and resolution of interface-level mismatches. We propose a method to identify parameters of mapping functions that resolve those mismatches. Second, we provide automated support for adapting behavioral models in presence of deadlock. Tackling this type of mismatch greatly expands the range of syntactically incompatible but adaptable services before automatically concluding that such services are not adaptable. We showed this using a number of examples and experiments (a case study) in the paper. In doing this, we exploit domain-specific knowledge available in the context of Web services, e.g., in WSDL interfaces, protocol specifications, and execution logs, if available, to provide above mentioned automated support.

We believe that these results are promising and encouraging. We have experienced that the problem is real and pressing, and the solution does considerably simplify adapter development.

References

[1] R. Akkiraju, A.-A. Ivan, R. Goodwin, S. Goh, and J. Lee. Semantic tools for web services. In *Emerging Technologies Toolkit (ETTK), IBM AlphaWorks*.

[2] BEA. Introduction to application integration. In *edocs.bea.com/wli/docs81/pdf/aiover.pdf*, June 2006.

[3] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani. Developing adapters for web services integration. In *Procs of CAiSE 2005*.

[4] B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data Knowl. Eng.*, 58(3), 2006.

[5] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *WWW'05*, 2005.

[6] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1), 2005.

[7] A. Brogi and R. Popescu. Automated generation of BPEL adapters. In *ICSOC'06*.

[8] N. Desai and M. P. Singh. Protocol-based business process modeling and enactment. In *IEEE International Conference on Web Services*, 2004.

[9] H. H. Do and E. Rahm. Coma - a system for flexible combination of schema matching approaches. In *VLDB*, 2002.

[10] X. Dong and et. al. Similarity search for web services. In *VLDB'04*.

[11] M. Dumas, M. Spork, and K. Wang. Adapt or perish: Algebra and visual notation for service interface adaptation. In *International Conference on Business Process Management*, 2006.

[12] M. Fuchs. Adapting web services in a heterogeneous environment. In *ICWS'4*, 2004.

[13] D. Hemer. A formal approach to component adaptation and composition. In *Australasian conference on Computer Science*, 2005.

[14] J. E. Hopcroft and J. D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[15] IBM. Websphere integration developer. In *www.ibm.com/software/integration/wid/*, 2006.

[16] W. Jiao and H. Mei. Automating integration of heterogeneous cots components. In *9th International Conference on Software Reuse*, 2006.

[17] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An aspect-oriented framework for service adaptation. In *ICSOC'06*.

[18] G. Kramler, E. Kapsammer, W. Retschitzegger, and G. Kappel. Towards using uml 2 for modelling web service collaboration protocols. In *Interoperability of Enterprise Software and Applications*, 2005.

[19] A. Martens and S. Moser. Diagnosing sca components using wombat. In *International Conference on Business Process Management*, 2006.

[20] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu, and A. K. Elmagarmid. Business-to-business interactions: issues and enabling technologies. *The VLDB Journal*, 12(1), 2003.

[21] A. A. Patil and et al. Meteor-s web service annotation framework. In *WWW'04*, 2004.

[22] S. Ponnekanti and A. Fox. Interoperability among independently evolving web services. In *Middleware'04*.

[23] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4), 2001.

[24] E. Rahm, H.-H. Do, and S. Mabmann. Matching large xml schemas. *SIGMOD Rec.*, 33(4), 2004.

[25] SCA. Service component architecture specifications. In *www.ibm.com/developerworks/library/specification/ws-sca.*

[26] W3C. Web services description language (WSDL) 1.1. In *www.w3.org/TR/wsdl*, 2001.

[27] Y. Wang and E. Stroulia. Flexible interface matching for web-service discovery. In *WISE'03*.

[28] S. K. Williams and et al. Protocol mediation for adaptation in semantic web services. *HP Technical Report HPL-2005-78*, 2005.

[29] X. Xiong and Z. Weishi. The current state of software component adaptation. In *First International Conference on Semantics, Knowledge and Grid*, 2005.

[30] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM TOPLAS*, 19(2), 1997.

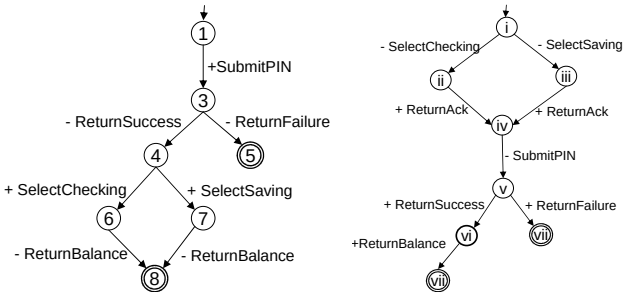
A ATM-Bank Example

As another example, let's consider the protocols of an ATM service (also referred as Bank) and client service (also referred to as ATM) as depicted in Figure 10(a), and Figure 10(b), respectively. The example has been highly simplified for presentation purposes, but it does exhibits some of the mismatches found in business protocols.

In this example, we assume that the user has already inserted the card into an ATM machine. Then the client selects the account type by sending a `SelectChecking` or `SelectSaving` message to the service. Then, it anticipates receiving message `ReturnAck`, before sending the message that contains user's PIN (Personal Identification Number). Following that the client needs to receive a notificaiton of (`ReturnSuccess`) or a failure message (`ReturnFailure`). On the other hand, the service anticipates to receive PIN information. Then, it issues a `ReturnSuccess` or `ReturnFailure`. According to the service's protocol, in case of `ReturnSuccess`, the client can select the type of account by sending message `SelectChecking` or `SelectSaving`. Finally, the service sends the balance accordingly.

These two protocols are not adaptable using available solutions in the literature. This is due to an ordering mismatches that results in deadlocks. The ordering mismatch relates to the fact that the client sends either `SelectChecking` or `SelectSaving` messages (which is of type of unspecified reception for the service), and is waiting to receive `ReturnAck`. However, the service is waiting to receive `SubmitPIN` first. Existing approaches create an empty adapter in this case, which means adaptation is not possible.

However, further analysis of these protocols reveals that these two protocols are in fact adaptable. This could be performed if `ReturnAck` message could be provided by the adapter, then the client will send `SubmitPIN`, and the interactions can go on successfully. Message `SelectChecking` (or `SelectSaving`) could be saved in the adapter to be forwarded to the service, when the service is in state 4 of the protocol.



(a) The ATM Service Protocol (b) ATM Client Protocol

Figure 10: The ATM Service and Client Protocols

B Applying the approach on ATM Example

The ATM service and client share the same interface definitions, except that there is no match for `ReturnAck` in the service interface. So, interface mappings in this case is straightforward for all other messages. As discussed before, for missing/extra messages, protocol-level analyze makes it clear if they are needed, and at what state during the interactions. In fact, in this particular case, the protocol-level analysis shows that this missing message leads the interactions of the two service to deadlock.

Figure 11 shows the mismatch tree MT for all interactions that result in two deadlocks between the two services (generated by Algorithm 5). The root node of MT is an *AND-Node*). One of the deadlocks occurs in state $\langle 1, ii \rangle$, and the other in state $\langle 1, iii \rangle$. In both cases, messages `TransmitPIN` and `ReturnAck` are responsible for the deadlock. This happens when ATM selects the account type (saving or checking) and waits for a reply, while the service is waiting to receive the PIN. Examining the WSDL interface of the ATM we find out that `ReturnAck` messages has the a string parameter called `account-type`, which can get one of the two values: `saving` or `checking` (defined as an enumeration inside the schema). In this case, this will get a probability 0.5 to construct it (one of the two possible values). As there is no log of previous communication of ATM is available (we are considering the compatibility of these two services), so it is not possible to specify which of the two values are used in which state. This is the reason that the probability is specified to be 0.5 instead of 1. On the other hand, the proposed mapping for `<Bank,SubmitPIN>` of the service is the message `<ATM,SubmitPIN>` in ATM client. However, it is not a plausible mapping, because at states $\langle 1, ii \rangle$, and $\langle 1, iii \rangle$ we have not received this message. The parameter of this message is a numerical value called `PIN`, for which no indication based on our evidences exists that it could be constructed, so it gets the probability of 0. In any of these two deadlock cases, if we assume that message `TransmitPIN` could be constructed, then two other deadlocks happen during the interactions of the two services as depicted in Figure 11, which shows that message `ReturnAck` is needed to resolve

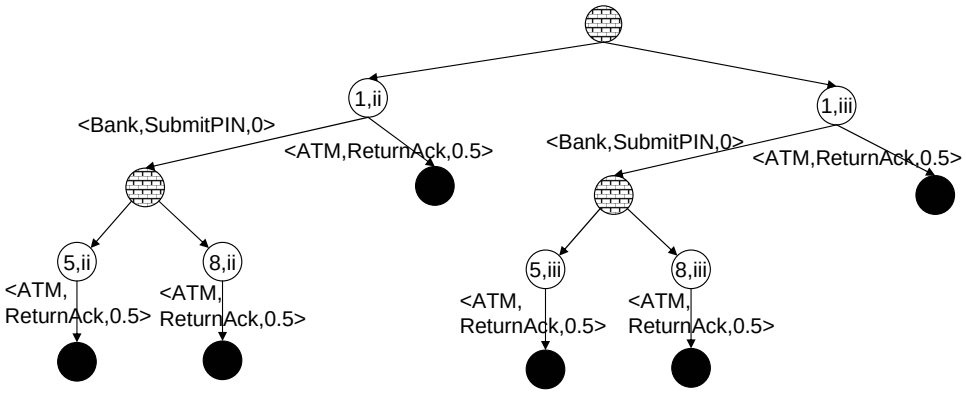


Figure 11: Weighted mismatch tree for the ATM service (Bank) and client (ATM) protocols

the deadlock eventually. However, both of such paths get a probability of zero as the probability of provision of `TransmitPIN` is zero.

Based on MT, the adapter developer confirms that is possible to construct `ReturnAck`. She specifies two mapping functions for message `ReturnAck`, one with the value `saving` and the other with the value `checking` for parameter `account-type`. The developer also defines the rules, via the rule editor in the tool (see Section C), to specify where to use each of these mappings. In fact, the first mapping should be used after receiving message `SelectChecking` by the adapter, or more precisely in state $\langle 1, ii \rangle$, and the next should be used after receiving message `SelectSaving` in state $\langle 1, iii \rangle$. Having these two mapping defined in the adapter, resolves both deadlock cases, and so makes it possible to generate the adapter specification for these two protocols.

C Tool Support

The implementation details of the tool has been discussed in Section 6. Figure 9 depicts the architecture of the tool. In this section, we provide some snapshots that shows the usage of the adaptation tool in the process of adapter generation for the ATM service and client services.

The ATM service and client are represented as SCA components inside an SCA module. The implementation of these two services is supplied as state machines (See Figure 12).

As discussed in Section B, providing the interface mappings are straightforward, and are implemented using *InterfaceMap* component in WID. The tool allows the adapter developer to select the two protocols and to check for protocol-level mismatches by simulating the adapter generation process (See left-hand side of Figure 12). Doing this, the adapter simulation/generation wizard will be executed (Figure 13). The simulation process ask for a file name for the generated

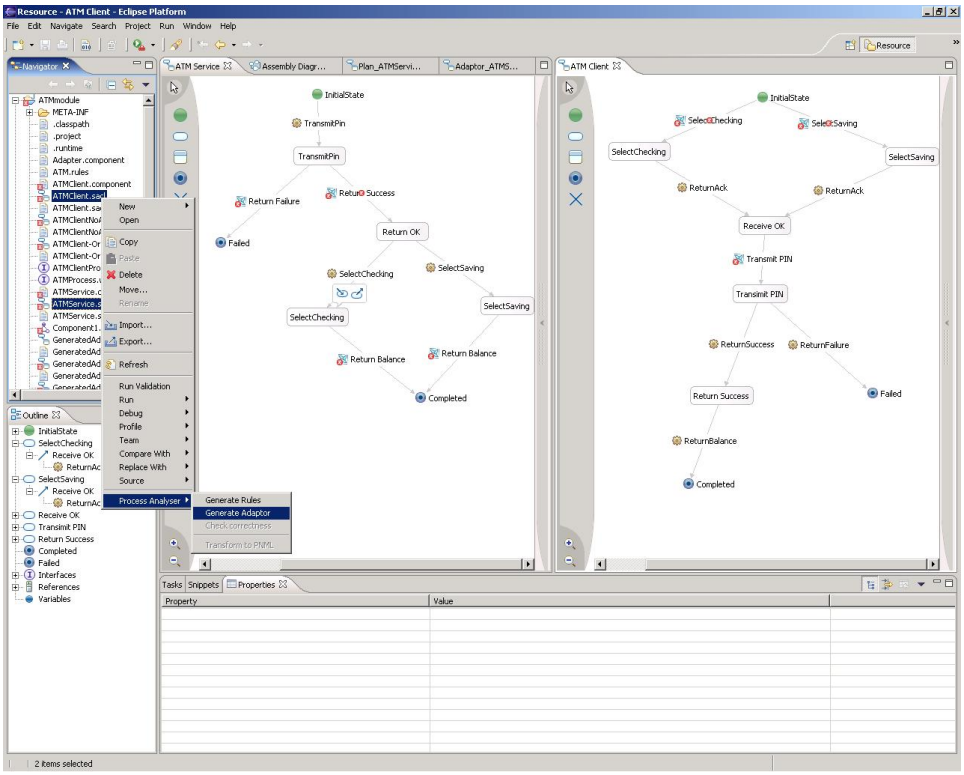


Figure 12: The protocols of ATM service and ATM client services

adapter, if possible to generate (having the default value of `GeneratedAdapter.sacl`). A file name for the mismatch tree is also requested, in case that there are deadlocks so the tree is generated (the default name is `PlanTree.sacl`.

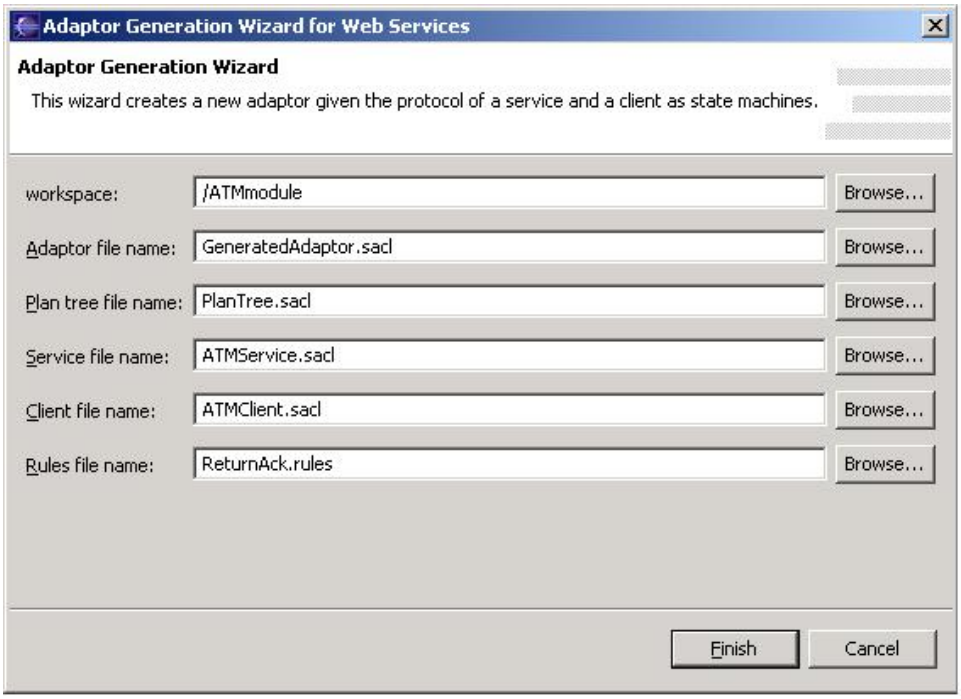


Figure 13: The adapter simulation wizard

Using this wizard, the developer may optionally provide the name of a file, in which contains user-defined rules. As discussed before, rules helps the adapter simulation/generation process to use the interface mappings at the right time. The result of adapter simulation is one of the followings:

- No adaptation is required, and the two process (services) can successfully interact.
- Adaptation required for successful interaction of services. However, there is no interactions leading to deadlock. Then the adapter specification is generated.
- Adaptation is required for successful interaction of services. There are interactions that lead to deadlock. Then, the adapter specification for all interactions without deadlocks is generated, and for interactions with the deadlock the mismatch tree is generated.

In this case, the tool specifies that there is no adapter between the two, as all interactions of the two services result in deadlocks. So, only the mismatch tree is generated. Figure 14 shows the mismatch tree for ATM service and client.

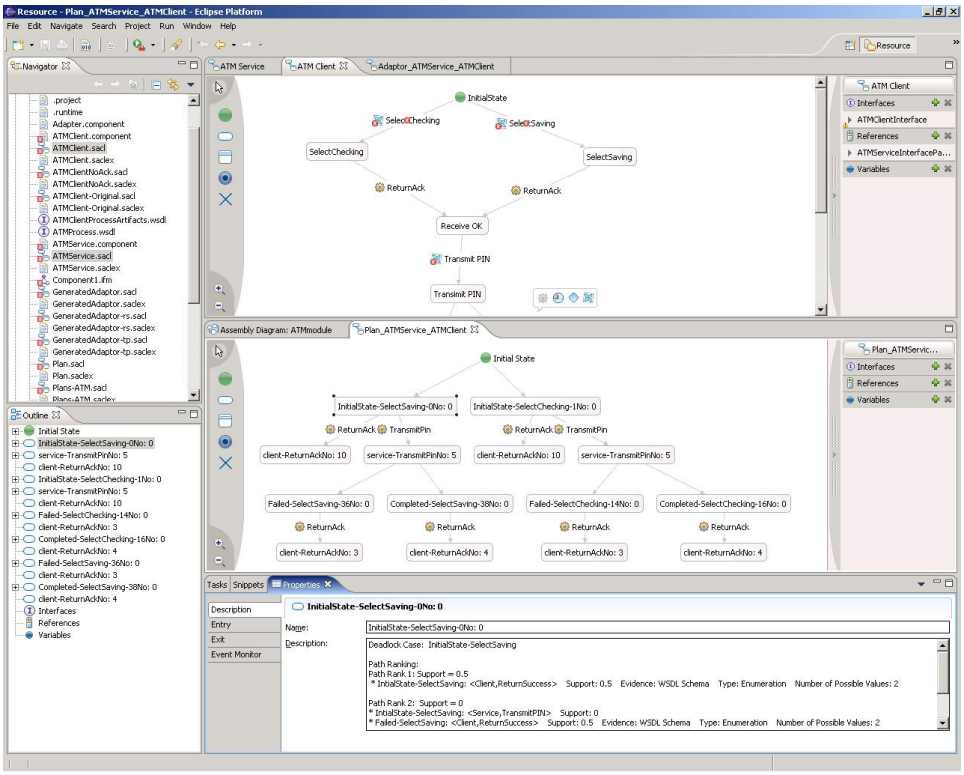


Figure 14: The mismatch tree for ATM service and client protocols

The state machine editor in WID is extended to visualize the mismatch tree. The weight of each edge in the tree is attached as properties of each transition. For each deadlock case, if the developer selects the root node for the deadlock tree, the suggestions for resolution of the deadlock are shown in the properties view (see Figure 14). In Figure 14, the deadlock that occurs in state $\langle InitialState - SelectSaving \rangle$ is selected. For this state, the tree path of the subtree belong to this deadlock case are ranked based on the probability that such messages could be constructed. Based on these results, the developer may define new interface mappings (e.g., in this case for construction of `ReturnAck` for each deadlock case). Finally, the tool allows the developer to define rules on how to use the interface mappings (See Figure 15). These rules are saved into rules files, which can be used as the input to the adapter generation process (see Figure 13).

Adapters are specified as SCA components that intermediates the interactions of ATM service and clients (See top window of Figure 16). In fact, an adapter component declares interfaces of both services so other services can call operations on the adapter, and also import the interfaces of the two services as *references*, so it can call operation's on the ATM service and client components. The implementation of an adapter is represented as state machines. In this case, after definition of required interface mappings for `ReturnAck` and definition of rules on where to use these mappings, the specification of the adapter state machines is generated by the tool according to definition 3.6 through adapter simulation/generation process. Figure 16 shows the adapter state machines that is generated for ATM service and client. In this case, the adapter generation process uses the rule file that is defined in Figure 15 as the input and the interface mappings generated for `<Client,ReturnSuccess>` to generate the adapter.

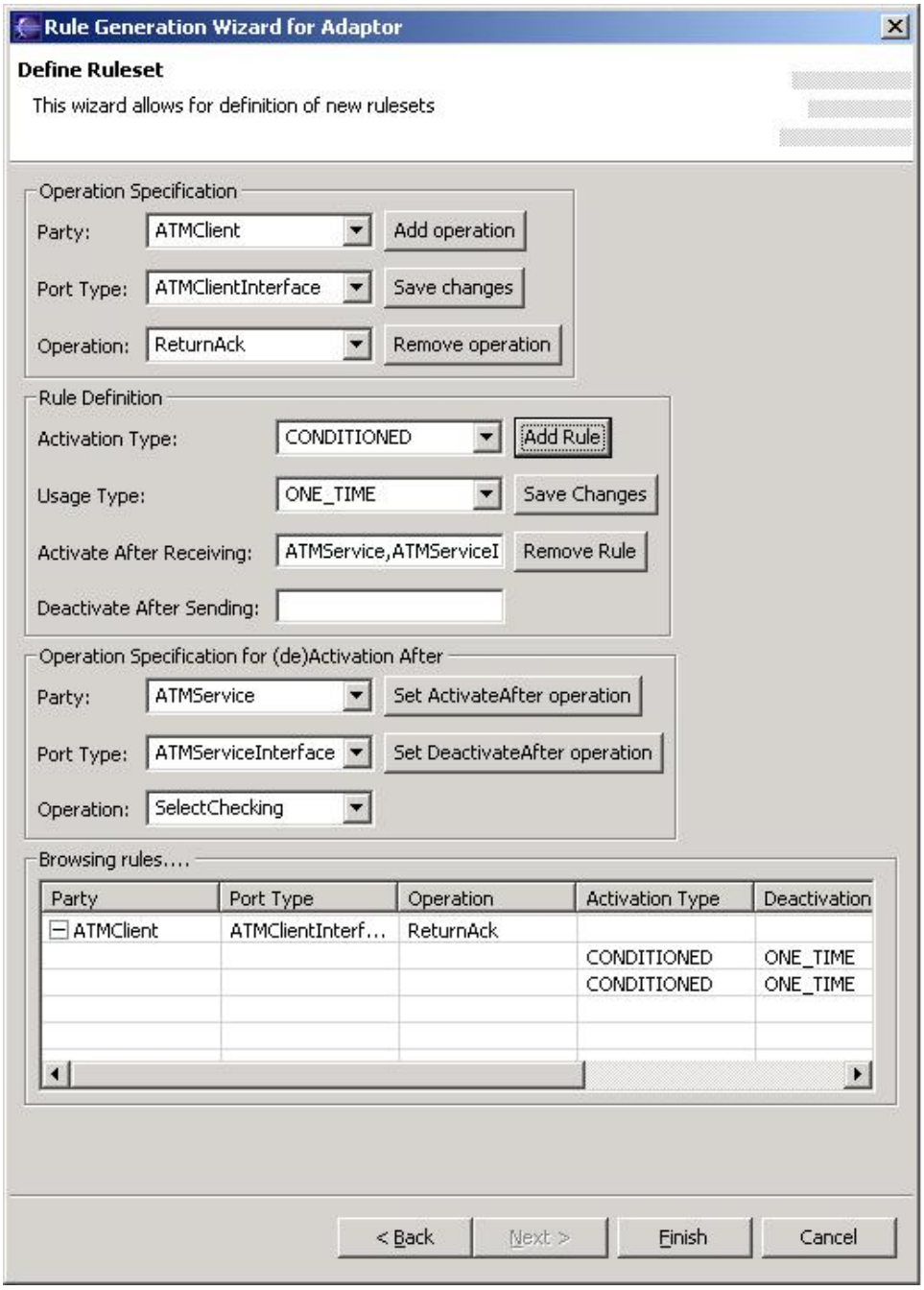


Figure 15: The snapshot of the rule definition editor

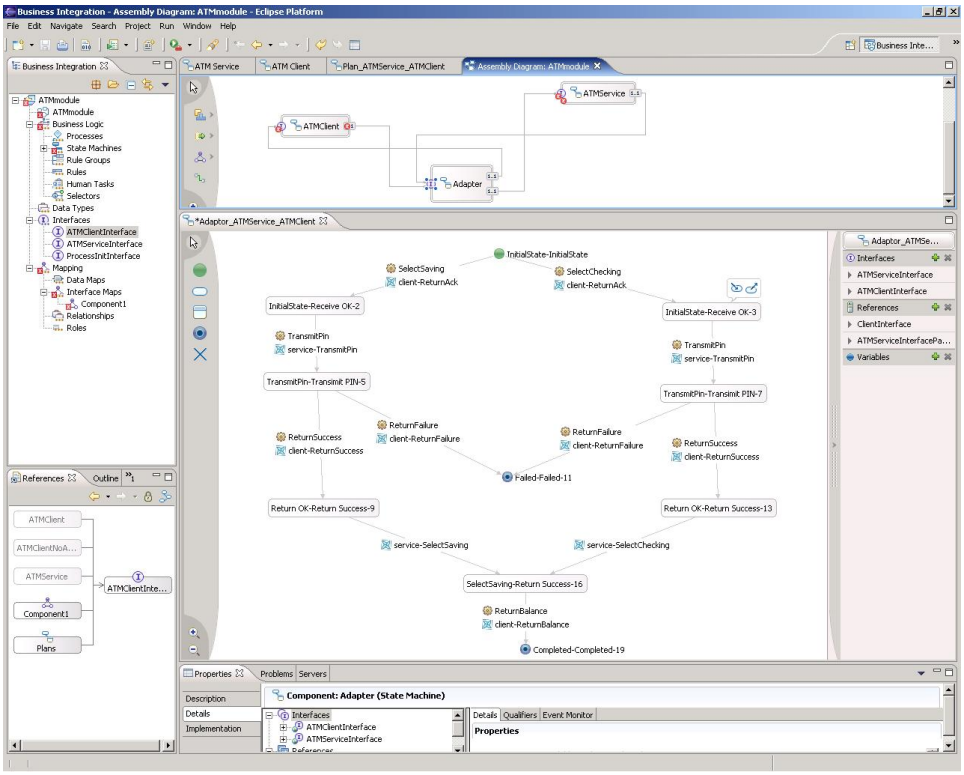


Figure 16: The protocol of the adapter for ATM service and client protocols